

## "CRACKING" DE CONTRASEÑAS

### Hashes: concepto, características y algoritmos

Las "funciones de resumen" son algoritmos matemáticos que consiguen crear a partir de una entrada cualquiera (ya sea el contenido de un archivo -binario o de texto, es igual- o una contraseña, por ejemplo) una **salida alfanumérica** de longitud **fija y** normalmente **menor** (el llamado "hash", o también "digest") que representa un resumen de toda la información que se le ha dado. Es decir, una determinada función de resumen, a partir de unos datos de entrada concretos, ha de crear una determinada cadena (el hash) que solo puede volverse a crear con esos mismos datos de entrada. Los hashes permiten, entre otras utilidades,:

\*Asegurar que no se ha modificado un archivo en una transmisión/descarga (es decir, verifican la integridad de un documento).

\*Hacer ilegible una contraseña (esto es para lo que los usaremos aquí)

Las funciones de resumen utilizadas en criptografía deben tener además una propiedad extra importante: la de asegurar que con el hash no se pueda saber nunca cuales han sido los datos insertados (es decir: estas funciones han de ser **unidireccionales**).

**NOTA:** This is a common confusion, especially because all these words are in the category of "cryptography", but it's important to understand the difference. **Encryption** transforms data from a cleartext to ciphertext **and back** (given the right keys), and the two texts should roughly correspond to each other in size: big cleartext yields big ciphertext, and so on. "Encryption" is a **two-way** operation. Hashes, on the other hand, compile a stream of data into a small digest (that is, a summarized form), and it's strictly a **one way operation**. All hashes of the same type have the same size no matter how big the inputs are.

De entre las funciones de resumen criptográficas más comunes podemos destacar los algoritmos: DES , MD5 , SHA-1 , SHA-2 (dentro del cual tenemos el SHA-224, el SHA-256, el SHA-384 y el SHA-512, según la longitud -en bits- del hash obtenido), RIPEMD (dentro del cual tenemos el RIPEMD-160 y el RIPEMD-320), WHIRPOOL, NTLM (usado en sistemas Windows), etc.. De todos ellos, el más difícil de revertir a día de hoy (y por tanto, el más seguro) es el SHA-512. En este sentido, el uso para tareas criptográficas de los algoritmos DES, MD5 y SHA-1 se desaconseja completamente.

En el caso concreto de las contraseñas, lo que se almacena siempre (ya sea en el archivo "/etc/shadow", en un servidor LDAP, en una base de datos, etc) es su hash, obtenido mediante una determinada función de resumen. De esta manera, en el caso hipotético de un robo de información, el atacante únicamente tendría los hashes de los usuarios, pero debería entonces intentar obtener a partir de ellos la contraseña respectiva, lo que puede ser una tarea imposible. En un caso típico de inicio de sesión, lo que ocurre es que a la contraseña introducida por el usuario el sistema aplica la misma función de resumen que con la que se generó el hash almacenado y seguidamente compara el hash recién obtenido con el almacenado: si son iguales, el acceso es concedido.

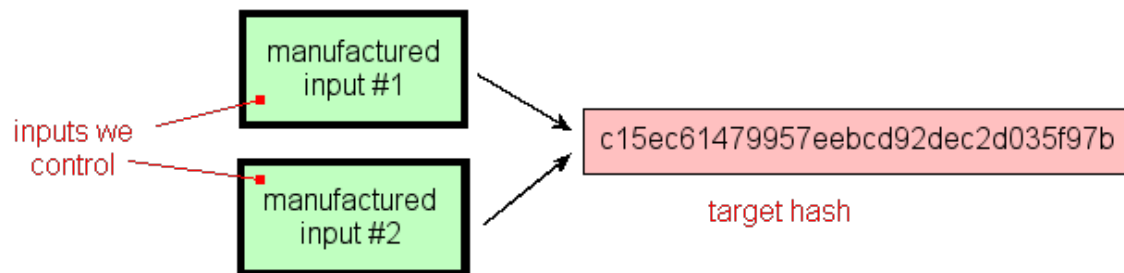
### Colisiones

Sabiendo que hay infinitas posibles entradas y solo un conjunto finito de posibles hashes, nos podemos preguntar si usando un determinado algoritmo dos entradas diferentes podrían generar dos hashes iguales y la respuesta es que teóricamente sí: por pura estadística podría haber **colisiones**:

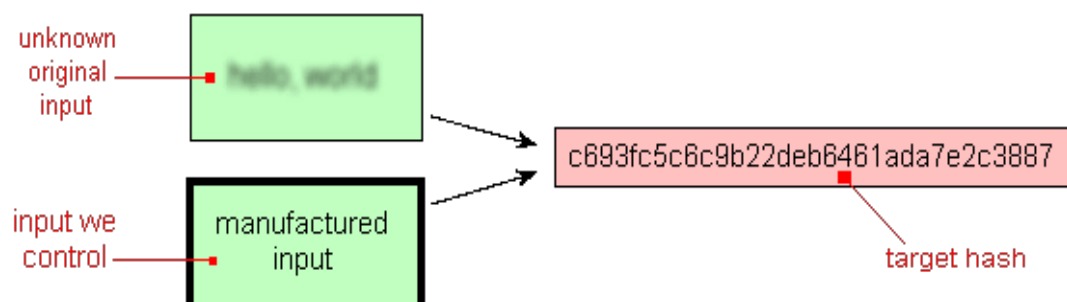
- El algoritmo MD5 genera un hash de 128 bits, por lo que hay "solo"  $2^{128}$  hashes diferentes posibles
- El algoritmo SHA-1 genera un hash de 160 bits, por lo que hay "solo"  $2^{160}$  hashes diferentes posibles
- El algoritmo SHA-256 genera un hash de 256 bits, por lo que hay "solo"  $2^{256}$  hashes diferentes posibles
- El algoritmo SHA-512 genera un hash de 512 bits, por lo que hay "solo"  $2^{512}$  hashes diferentes posibles

No obstante, si el algoritmo está bien diseñado y distribuye los hashes uniformemente en todo el espectro de salida, encontrar una colisión al azar será extremadamente poco probable. Esto es importante porque collisions play a central role in the usefulness of a cryptographic hash in the sense that the easier it is to find a collision, the less useful the hash is. Some algorithms are better than others at avoiding collisions, and this is measured by three related attributes.

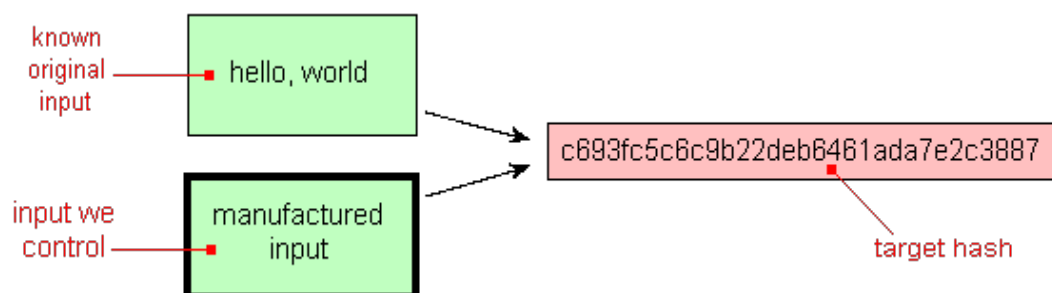
*\*Collision Resistance:* Measures how difficult it is to pick two inputs that produce the same hash value. We manufacture both of the inputs in an attempt to coax the same hash value from each, and we don't care what the *particular* hash value generated is (just that they both match).



*\*Preimage Resistance:* Measures how difficult it is to concoct an input which hashes to a particular value. If we are able to "work backwards" from a hash and create some text that produces the same hash, we can use this to beat hashed passwords. We won't ever know the actual input data that was used, but that doesn't matter. Looking at the flow for validating against hashed passwords, all that matters is that the hashes match, not the passwords, so if we can find *any* other text that produces the stored hash, we'll be granted access. "Collisions" mean "more than one password will be accepted".



*\*Second Preimage Resistance :* Measures how difficult it is to concoct an input which hashes to the same value that some other given input hashes to. the goal is to produce a new input that generates the given hash, but this time we know the original text that created it. However, it's not clear just how much help this extra knowledge is.



## Hashes en la práctica

El comando *md5sum* permite generar un hash MD5 de los datos introducidos interactivamente por teclado (cuya entrada ha de finalizarse con CTRL+D) o bien del contenido de un fichero si su ruta se indica como parámetro. Igualmente, tenemos los comandos *sha256sum* o *sha512sum* que funcionan de forma similar. Alternativamente también podemos usar para esto la suite criptográfica generalista *openssl*, así (si se emplea el teclado como entrada de datos en vez de un fichero hay que pulsar CTRL+D tres veces):

```
openssl md5 [/ruta/fichero] o bien openssl dgst -md5 [/ruta/fichero]
openssl sha512 [/ruta/fichero] o bien openssl dgst -sha512 [/ruta/fichero]
...
```

Los comandos anteriores nos serán útiles cuando queramos verificar la integridad de algún archivo descargado de Internet (por ejemplo, una iso o un programa cualquiera como por ejemplo Gimp) comparando el hash obtenido con el hash que el proveedor de ese archivo muestre en su sitio web. No obstante, para "encriptar" contraseñas de usuarios de un sistema Linux no nos servirán. ¿Por qué? Porque a pesar de que en las distribuciones Linux modernas se utiliza el algoritmo SHA512 (esto se puede confirmar que es así efectivamente observando el valor de la línea ENCRYPT\_METHOD presente en el archivo /etc/login.defs), el proceso de "encriptación" es más complejo.

**NOTA:** Hay que tener en cuenta que la línea ENCRYPT\_METHOD mencionada en el párrafo anterior indica el algoritmo a usar por defecto para "hashear" una nueva contraseña siempre y cuando no se haya indicado ningún otro en la línea "password sufficient pam\_unix.so ..." del archivo /etc/pam.d/common-password (en Ubuntu) o /etc/pam.d/system-password y /etc/pam.d/system-auth (en Fedora). Sobre estos archivos hablaremos en estudiar el subsistema PAM

En concreto, lo que ocurre es que la función resumen no se realiza sobre la contraseña solamente sino sobre la pareja sal+contraseña. La **sal** es un número aleatorio de longitud fija (normalmente 16 bits), también secreto y, sobre todo, único, que el sistema genera y añade a la contraseña para entonces calcular el hash de la combinación. En este sentido, a la hora de comprobar el correcto inicio de sesión de un usuario, lo que ocurre en realidad es lo siguiente:

- 1.-Se recupera la sal utilizada en la generación del hash (que ha de estar, obviamente, almacenada en un lugar conocido por el sistema)
- 2.-Se añade esa sal a la contraseña introducida por el usuario y se calcula el hash resultante
- 3.-Se compara ese hash con el almacenado por el sistema. Si coinciden, se concede acceso

La razón de ser de la sal es para evitar ataques contra contraseñas mediante hashes precalculados. Es decir, la mayor parte del tiempo empleado en adivinar una contraseña se emplea en el cálculo del hash de cada contraseña candidata: si ese cálculo ya se ha hecho previamente y lo que el atacante tiene es simplemente una base de datos inmensa de hashes asociados a su contraseña correspondiente, la adivinación se reduce a encontrar dentro de esa base de datos el hash adecuado. Con el añadido de la sal, este tipo de ataque por "base de datos de hashes" (un nombre que se les da es el de "tablas Rainbow") se hace imposible porque se debería tener precalculado, para cada contraseña candidata, tanto hashes como diferentes sales podrían existir (recordemos, números aleatorios de normalmente 16 bits, por lo que salen  $2^{16}$  sales diferentes), lo que lo hace totalmente inviable a nivel de almacenamiento debido a la cantidad inmensa de hashes que resultan. Por otro lado, otra ventaja muy importante de usar sales es que dos usuarios pueden tener la misma contraseña pero el hash resultante es diferente debido a que la sal es diferente para cada uno de ellos (porque se regenera "insitu" cada vez que se actualiza una contraseña particular).

En sistemas Linux los hashes sal+contraseña se almacenan en el segundo campo del archivo "/etc/shadow". No obstante, hay que tener en cuenta que ese segundo campo tiene el siguiente "formato"...

*\$algoritmoHash\$sal\$hashsalycontraseña*

...donde entre los dos primeros símbolos "\$" lo que aparece es una marca indicando el algoritmo utilizado para generar el hash (esta marca suele valer 6, lo que indica que el hash es SHA-512; otros valores pueden ser 5 para SHA-256) o 1 para MD5; para saber más valores consultar *man 3 crypt* -en Ubuntu- o *man 5 crypt* -en Fedora-) y donde entre el segundo y tercer "\$" se almacena la sal empleada para generar ese hash en

concreto, el cual, entonces sí, se guarda tras el tercer "\$" . Para generar un hash SHA-512 a partir de una combinación sal+contraseña dada que sea válido para ser almacenado en el archivo "/etc/shadow" (al tener el formato necesario \$6\$valor\$hashconjunto) podemos ejecutar el siguiente comando:

*openssl passwd -6 -salt valor\$al contraseña*

**NOTA:** Alternativamente al comando anterior también podríamos emplear pequeños scripts en Perl o Python. En concreto, en Python podemos hacer: `python3 -c 'import crypt; print(crypt.crypt("hola",crypt.mksalt(crypt.METHOD_SHA512)))'` , donde "hola" representa la contraseña a "hashear" con una sal SHA-512 aleatoria. Si se quisiera indicar una sal SHA512 concreta, se debería escribir así: `python3 -c 'import crypt; print(crypt.crypt("hola","$6$salConcreta"))'` Para más información sobre cómo generar sales y contraseñas adecuadas, leer <https://crackstation.net/hashing-security.htm>

También nos podríamos encontrar con la situación contraria: disponer de un hash determinado y no saber con qué algoritmo ha sido generado. En este caso, nos puede servir un script Python descargable desde <https://github.com/blackploit/hash-identifier> o bien alguna herramienta online como [https://md5hashing.net/hash\\_type\\_checker](https://md5hashing.net/hash_type_checker) También podemos observar la lista de hashes de ejemplo según el algoritmo utilizado que se muestra en [https://hashcat.net/wiki/doku.php?id=example\\_hashes](https://hashcat.net/wiki/doku.php?id=example_hashes) para comparar su "forma" con el hash que tengamos (o ejecutar el comando *hashcat --example-hashes*, ver apartado siguiente). De esta manera, una vez reconocido el algoritmo usado, ya podríamos empezar el proceso de "crackeo".

## Hashcat

El programa Hashcat (<https://hashcat.net/hashcat>) serveix per "crackejar" hashes. Tant a l'Ubuntu com a Fedora es pot trobar als repositoris oficials. Amb aquest programa podem fer servir bàsicament dos mètodes d'adivinació o "modes d'atac":

\*Per "força bruta": Genera múltiples contrasenyes "candidates" una darrera una altra, calcula el "hash" corresponent de cadascuna d'elles i els compara un a un amb el hash que volem "crackejar" (obtingut prèviament d'alguna determinada manera). Tot això en temps real.

**NOTA:** En aquest tipus d'atac és evident que quantes més combinacions de contrasenyes hagi de provar el "crackejador" més difícil ho tindrà per trobar-ne la bona. Per mesurar la dificultat d'esbrinar una contrasenya es fa servir una fórmula matemàtica que calcula la quantitat de variacions possibles donat un conjunt de caràcters concret (amb repeticions) en un número concret de posicions. Aquesta fórmula és:  $x^y$  , on "x" representa la quantitat de caràcters i "y" el número de posicions. Per exemple, si sabem que la contrasenya només té lletres minúscules i té una longitud de quatre lletres,  $x=26$  i  $y=4$ . Si té igual quatre lletres però aquestes poden ser ara minúscules o majúscules, llavors  $x=52$  i  $y=4$  , etc. Es veu clarament, doncs, que el número de combinacions possibles pot augmentar per dos motius: o bé augmentant la quantitat de caràcters que poden aparèixer a la contrasenya i/o bé augmentant la seva longitud. És per això que s'aconsella en general tenir contrasenyes amb lletres minúscules, majúscules, xifres, símbols, etc i que siguin de vuit caràcter com a mínim...la quantitat de temps necessari per tal de provar totes les combinacions possibles si es fa així és major que l'edat de l'univers. De totes formes, cal dir que amb l'ús de "màscares" (de seguida en parlarem) el número de combinacions a provar es redueix dràsticament, així que tampoc podem estar segurs del tot.

\*Per "diccionari": Els diccionaris són simplement fitxers de text que contenen llistes de paraules, cadascuna escrita en una línia diferent. Aquestes paraules representen les contrasenyes "candidates" concretes de les quals s'anirà calculant el hash corresponent per tal de comparar-lo amb el hash que volem "crackejar". Més endavant veurem d'on podem obtenir diccionaris ja fets o fins i tot, com els podem construir nosaltres.

**NOTA:** Per afegir efectivitat a un atac per "diccionari" es poden utilitzar "regles". Les regles són combinacions predefinides de lletres i números que es poden aplicar a les paraules d'un diccionari per tal d'ampliar les variacions d'aquestes. D'aquesta manera no cal tenir escrites aquestes variacions explícitament dins del diccionari (cosa que el faria molt llarg i feixuc) sinó que es poden generar en temps real. Per exemple, una regla podria ser "passar a minúscula tots els caràcters de les paraules del diccionari"; una altra altra podria ser "afegir al final de totes les paraules del diccionari el número 123", etc, etc.

El mètode per força bruta sempre acabarà trobant la contrasenya, però pot trigar centenars d'anys (tot i que amb l'ús de "màscares", com veurem de seguida, aquest temps es pot reduir); el mètode per diccionari és més ràpid però està limitat a les contrasenyes que hi contingui aquest.

Un cop instal·lat, si es vol realitzar un atac per diccionari d'un hash típic d'usuari Linux, el seu funcionament general és així:

```
hashcat -m 1800 -a 0 /ruta/arxiuAmbHash.txt /ruta/arxiuDiccionari.txt
```

**-m** : Indica el tipus de hash a crackejar. El valor 1800 concretament significa SHA-512. Altres valors es poden conèixer executant *hashcat -h* (o a <https://hashcat.net/wiki/doku.php?id=hashcat>)

**-a** : Indica el tipus d'atac. El valor 0 indica per diccionari. Altres valors es poden conèixer executant també *hashcat -h* (o a <https://hashcat.net/wiki/doku.php?id=hashcat>)

Cal tenir en compte que el fitxer on es troba el hash a "crackejar" només pot contenir hashes (més d'un si es vol, un per cada línia, però només hashes); per tant, cal eliminar primer els altres camps que apareixen en un arxiu "/etc/shadow" típic. D'altra banda, l'ordre per indicar les rutes dels fitxers amb el hash i del fitxer de diccionari ha de ser aquest, no al revés.

Opcionalment, després de la ruta de l'arxiu de diccionari es pot afegir el paràmetre *-r /ruta/arxiu.regles* per tal d'aplicar sobre el diccionari en qüestió totes les regles escrites dins l'arxiu indicat. Per defecte Hashcat proporciona uns quants arxius amb extensió ".rule" dins de la carpeta "/usr/share/hashcat/rules" (a Ubuntu) o "/usr/share/doc/hashcat-doc/rules" (a Fedora), els quals també estan disponibles per descarregar individualment a <https://github.com/hashcat/hashcat/tree/master/rules> (altres regles de tercers es poden obtenir per exemple a <https://github.com/ptraetorian-inc/Hob0Rules> o <https://github.com/NSAKEY/nsa-rules> o <http://contest-2010.korelogic.com/rules-hashcat.html> , entre altres).

La sintaxi per construir regles pròpies es pot consultar a [https://hashcat.net/wiki/doku.php?id=rule\\_based\\_attack](https://hashcat.net/wiki/doku.php?id=rule_based_attack) (encara que també podríem aprofitar aquesta pàgina més amigable: <https://www.4armed.com/blog/hashcat-rule-based-attack> ). De totes formes, a continuació es llisten algunes de les regles més senzilles:

:	Prova cada paraula del diccionari tal qual, sense no realitzar cap canvi a com està escrita
<i>l o u</i>	Passa a minúscula o majúscula totes les lletres de cada paraula del diccionari, respectivament
<i>c</i>	Posa en majúscula la primera lletra i en minúscula la resta de lletres de cada paraula del diccionari
<i>C</i>	Posa en minúscula la primera lletra i en majúscula la resta de lletres de cada paraula del diccionari
<i>t</i>	Posa a minúscula les lletres que estan a majúscula (i viceversa) de cada paraula del diccionari
<i>TN</i>	Posa a minúscula la lletra que ocupa la posició N si està a majúscula (o viceversa) de cada paraula
<i>\$X o ^X</i>	Afegeix "X" al final o al principi de cada paraula del diccionari, respectivament
<i>sXY</i>	Substitueix totes les ocurrences de "X" amb "Y" a cada paraula del diccionari
<i>DN</i>	Elimina la lletra que ocupa la posició N de cada paraula del diccionari
<i>@X</i>	Elimina "X" de cada paraula del diccionari
<i>d</i>	Genera, per cada paraula del diccionari, una paraula fruit de la seva duplicació ("hola"->"holahola")
<i>pN</i>	Afegeix N vegades darrera de cada paraula del diccionari ella mateixa
<i>zN o ZN</i>	Duplica la primera o darrera lletra de cada paraula del diccionari, respectivament
<i>r</i>	Inverteix cada paraula del diccionari
<i>f</i>	Afegeix al final de cada paraula del diccionari la seva inversió
<i>{ o }</i>	Mou una lletra de cada paraula del diccionari cap a l'esquerra o la dreta, respectivament
<i>[ o ]</i>	Elimina la primera lletra de cada paraula del diccionari o l'última, respectivament

**NOTA:** Hi ha més exemples de regles senzilles aquí <http://openwall.info/wiki/media/john/korelogic-rules-20100801.txt> o al mateix repositori oficial de Hashcat (<https://github.com/hashcat/hashcat/blob/master/rules> ) on podem trobar diferents fitxers de regles distribuïts amb el programa com ara "best64.rule", "leetspeak.rule" o "combinator.rule", entre d'altres

**NOTA:** Si es volen aplicar vàries regles en cadena escrites en diferents fitxers (és a dir, primer aplicar una regla escrita en un fitxer i sobre el resultat obtingut aplicar-ne una altra escrita en un altre fitxer), es poden indicar els diferents arxius de regles implicats cadascun amb un paràmetre `-r` diferent. L'ordre en què s'indiquin és important per definir l'ordre de la cadena.

**NOTA:** Per conèixer el grau d'efectivitat de les regles, es poden afegir els paràmetres `--debug-mode=1` i `--debug-file=/ruta/fitxer`, els quals activaran la gravació dins del fitxer indicat de les estadístiques d'efectivitat de les diferents variants de contrasenyes provades

---

Si es vol realitzar un atac per força bruta, a Hashcat només es pot fer indicant "màscara". El seu funcionament general és així:

```
hashcat -m 1800 -a 3 /ruta/arxiuAmbHash.txt mascara
```

on ara l'atac passa a ser el "3" i la "màscara" indica la forma que pensem que té la contrasenya a buscar. Concretament, una màscara per exemple que sigui `?d?d?d` només provarà contrasenyes que tinguin exactament tres números; una altra que sigui `?u?!?!?` provarà contrasenyes de quatre lletres on la primera sigui majúscula i la resta minúscules, i així. Els valors possibles per generar màscares es poden consultar veient la sortida de `hashcat -h` però bàsicament són:

```
?l : abcdefghijklmnopqrstuvwxyz
?u: ABCDEFGHIJKLMNOPQRSTUVWXYZ
?d: 0123456789
?s: !"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
?a: ?l?u?d?s
?h: 0123456789abcdef
?H: 0123456789ABCDEF
```

**NOTA:** Dins de la màscara es pot incloure contingut literal. Per exemple, si volguéssim buscar contrasenyes que comencessin per una lletra majúscula, després tinguessin el número "19" i després una lletra minúscula, es podria fer servir la màscara `?u19?l`

Encara que la màscara digui per exemple `?a?a?a?a` es pot no haver de comprovar contrasenyes que tinguin exactament 5 caràcters sinó que es pot anar incrementant el número a provar pas a pas (primer contrasenyes amb un caràcter començant per l'esquerra, després amb dos, després amb tres, etc) des d'un mínim fins un màxim. Per fer això s'ha d'afegir els paràmetres `-i --increment-min=nº` o bé `-i --increment-max=nº` (o tots dos).

També és possible definir conjunts nous de caràcters mitjançant els paràmetres `-1`, `-2`, `-3` o `-4`. Per exemple, si volem definir un conjunt format per tots els dígitos numèrics i totes les lletres minúscules, un altre grup format per tots els dígitos numèrics i (només) les lletres a, b, c i d i un altre grup format per totes les lletres minúscules, totes les majúscules i (només) els dígitos 0,1,2 i 3, podríem afegir els paràmetres `-1=?d?l`, `-2=?dabcd` i `-3=?l?u0123` i llavors escriure com a màscara el següent, per exemple: `?1?3?2?1?d`.

**NOTA:** Tingueu en compte que a la definició del conjunt només estem indicant el tipus de caràcter admés, no pas cap ordre dels caràcters que el formen (és a dir, si per exemple tenim `-1=?s?d` i fem servir una màscara tal com `?1?1`, estarem buscant contrasenyes de només dos caràcters que poden ser símbols o decimals tots dos indistintament)

Un altre atac que podem fer servir és el de combinació, que consisteix en tenir dos diccionaris i provar contrasenyes que són el resultat de combinar cadascuna de les paraules del primer amb totes les del segon. S'hauria de realitzar així:

```
hashcat -m 1800 -a 1 /ruta/arxiuAmbHash.txt /ruta/arxiuDic1.txt /ruta/arxiuDic2.txt
```

En la comanda anterior es pot aplicar una regla concreta al diccionari de l'esquerra afegint el paràmetre `-j "regla"` i/o una altra regla al diccionari de la dreta amb el paràmetre `-k "regla"`.

També estan els modes híbrids: diccionari + màscara (atac nº6) o bé màscara + diccionari (atac nº7). Per exemple, el primer cas s'executaria així:

```
hashcat -m 1800 -a 6 /ruta/arxiuAmbHash.txt /ruta/arxiuDic1.txt mascara -i
```

Per més informació podeu llegir [https://hashcat.net/wiki/doku.php?id=hybrid\\_attack](https://hashcat.net/wiki/doku.php?id=hybrid_attack)

Hashcat manté l'arxiu "~/.hashcat/hashcat.potfile" contenint els diferents resultats amb èxit obtinguts al llarg de la història de la seva execució. D'aquesta manera, executant `hashcat -m 1800 --show ~/.hashcat/hashcat.potfile` podem tornar a saber les contrasenyes que tenim endevinades provinents de hashes de tipus SHA-512. Es pot deshabilitar completament aquest arxiu (això faria que Hashcat intentés de nou "crackejar" contrasenyes ja conegudes) si afegim el paràmetre `--potfile-disable` en el moment de realitzar l'atac; també es pot definir un altre fitxer diferent amb el paràmetre `--potfile-path /ruta/fitxer`

Altres paràmetres interessants de Hashcat són:

- `--remove` : esborra els hashes de l'"arxiuAmbHash.txt" a mesura que es van endevinant
- `--username` : indica si el primer camp de cada línia d'"arxiuAmbHash.txt" és el nom de l'usuari (del tipus "*usuari:hash*", molt habitual) per tal d'interpretar correctament el seu contingut
- `--stdout` : *mostra* a pantalla les combinacions resultats d'aplicar un fitxer de regles sobre un determinat diccionari. Si s'usa aquest paràmetre llavors només caldria indicar el diccionari i el fitxer de regles, així: `hashcat dict.txt -r regles.rule --stdout` (ja que la idea no és crackejar res sinó simplement computar variacions de caràcters).
- `-o /ruta/arxiuResultat.txt` : Indica la ruta on es guardarà la contrasenya "crackejada". Si aquest paràmetre no s'escriu, la contrasenya trobada es mostrarà a pantalla i es guardarà al potfile
- `--status` : Mostra per pantalla l'estat del "crackeig" actualitzat-se automàticament
- `--restore nom` : Si prèviament s'ha executat hashcat amb el paràmetre `--session nom`, restaura la sessió de "crackeig" indicada per continuar per on es va quedar la darrera vegada
- `--left` : Contrari a `--show`: mostra els hashes que encara no s'han "crackejat" de l'"arxiuAmbHash.txt" indicat
- `-O` : It greatly increases the cracking speed, although limits the password length that you'll be able to crack. This is usually fine, unless you are cracking passwords greater than 27 characters.
- `--cpu-affinity=1,2,3` : Associa l'execució del procés hashcat a un nucli de CPU concret
- `-n nº` , `-u nº` , `-T nº` : Diferents opcions d'optimització en relació als threads utilitzats

## Wordlists

Existeixen molts diccionaris disponibles a Internet per descarregar. A més dels propis paquets de cada distribució (un simple *apt search wordlist* ens pot servir per donar una ullada, o fins i tot un *locate wordlist*) tenim aquests altres:

<https://github.com/berzerk0/Probable-Wordlists>  
<https://weakpass.com>  
<https://github.com/danielmiessler/SecLists/tree/master/Passwords>  
<https://wordlists.capsop.com>  
<http://human0id.net/wordlists.html>  
<https://wiki.skullsecurity.org/Passwords>  
<http://downloads.skullsecurity.org/passwords/500-worst-passwords.txt.bz2>  
<http://downloads.skullsecurity.org/passwords/twitter-banned.txt.bz2>  
<http://downloads.skullsecurity.org/passwords/john.txt.bz2>  
<http://downloads.skullsecurity.org/passwords/cain.txt.bz2>  
<https://packetstormsecurity.com/Crackers/wordlists/>  
<http://mirrors.kernel.org/openwall/wordlists/>  
<https://crackstation.net/buy-crackstation-wordlist-password-cracking-dictionary.htm>  
<http://finder.insidepro.com> (buscador de hashes en base de dades online)



Un article molt interessant que compara diferents wordlist per esbrinar quin és el més complet, net, no redundant, etc és <https://blog.g0tmi1k.com/2011/06/dictionaries-wordlists>

D'altra banda, també existeix la possibilitat de generar-se un mateix el seu propi diccionari. De fet, hi ha eines que automatitzen aquest procés com ara <https://github.com/Mebus/cupp> , <https://sourceforge.net/projects/crunch-wordlist> o <https://github.com/berzerk0/BEWGor>

També són interessants les llistes de contrasenyes per defecte de dispositius (generalment "routers") de diferents fabricants, com ara <http://www.defaultpassword.com> o <https://cirt.net/passwords>

## EXERCICIS:

**1.-a)** Crea un fitxer de text amb algun contingut a dins i genera seguidament el seu hash SHA512 amb la comanda *sha512sum* . A continuació, modifica el seu contingut d'alguna manera i torna a generar un nou hash. ¿És igual a l'anterior? ¿Quants caràcters té el hash generat en ambdós casos?

**b)** Utilitza la comanda *openssl passwd ...* explicada a teoria per generar un hash SHA512 per la contrasenya "usuari" saltejat amb la sal "1a2b3c". Ara comprova si el hash que has obtingut és igual al que hi ha dins de l'arxiu "/etc/shadow". Per què? ¿Com podries fer que sí fossin iguals?

**NOTA:** Una aplicació més especialitzada i versàtil que *openssl* relacionada amb la creació i comprovació de hash de diferents algoritmes és <https://github.com/ColumPaget/Hashrat> , per si la voleu provar

**2.-a)** Visita <https://hashes.org/leaks.php#> i digues què significa la llista que allà mostra. ¿Per a què serveix el botó "Download found" i el botó "Download leftlist"?

**b)** Fes servir <https://hashes.org/search.php> per esbrinar si el hash MD5 "00000be2159d84c848ea79a3fd6a3ac7" ha sigut crackejat. També es podries fer servir <https://hashes.org/api.php> per fer la recerca.

**c)** A <https://hashes.org/gen.php> podem trobar un generador de hash online. Llegeix primer l'ajuda a <https://hashes.org/algorithms.php> (o també <https://s3inlc.wordpress.com/2017/11/10/algorithms-on-hashes-org>) per esbrinar quina diferència hi ha entre especificar a la caixa de text del formulari titulada "Hash algorithm" el valor "SHA1" o el valor "SHA1x2" (per qualsevol entrada)

Instal·la ara a la màquina virtual el paquet "hashcat". ¡Atenció, però! Degut a què Hashcat utilitza la GPU de la màquina i el VirtualBox no emula cap dispositiu d'aquest tipus, si proves d'executar qualsevol atac dels vistos a la teoria amb Hashcat, aquest es queixarà de què no troba cap GPU i es parará (aquest fet es pot comprovar executant la comanda *hashcat -I*, que serveix per informar de les eventuais GPU detectades).

Per evitar aquest inconvenient, un "arranjament" és indicar-li explícitament que utilitzi la CPU, la qual, de fet, és l'únic dispositiu hardware que no està virtualitzat. Això farà que vagi molt més lent (les GPUs tenen una estructura interna que resulta ser més adient per paral·lelitzar tasques com la generació i comparació de hash que la que tenen les CPUs) però com a mínim funcionarà.

**NOTA:** Per tal de fer que la CPU pugui simular per software actuar com si fos una GPU primer caldrà tenir instal·lat a la màquina virtual el paquet "**pocl-openc1-icd**" (a Ubuntu) o els paquets "**pocl**", "**ocl-icd**", "**opencl-headers**" i "**clang**" (a Fedora). Afortunadament, la instal·lació de Hashcat via repositoris oficials ja haurà instal·lat aquests paquets com a dependències.

A partir d'aquí, al Hashcat caldrà indicar sempre el paràmetre *--force* per evitar què el Hashcat no s'aturi en no detectar cap GPU i el paràmetre *-D 1* per indicar explícitament que es farà servir la CPU i no pas cap GPU (encara que si no hi ha res més per escollir que la CPU, Hashcat ja la seleccionarà automàticament).



3.-Un hacker rus amic teu ha aconseguit obtenir la següent línia de l'arxiu "/etc/shadow" del servidor principal de la central nuclear de Kamchatka i et demana que esbrinis la contrasenya corresponent...:

```
pepe:$6$7zeqI7nh4zBpJ59j$wj0KejQ5mR3UXsfglmOdoiDhNlkHk/  
h9UmTgUoVa9mXSduup7k4bpGYtjuYTqG1n230vXgeYLCCbQ.bEuN8KM1:17444:0:99999:7:::
```

a) ...utilitzant Hashcat i el mètode per diccionari sense regles (descarrega't el diccionari que vulguis de la llista d'enllaços mostrada a la teoria). Si arribessis a esbrinar la contrasenya (que haurà de sortir impresa a pantalla juntament el seu hash associat), elimina el potfile per poder començar de nou al següent exercici.

**NOTA:** Recorda que mentre es va realitzant la recerca pots pulsar la tecla "s" per tal d'observar el tems estimat de finalització, -així com altres dades d'interès com ara les combinacions concretes provades en aquell moment, el número màxim de combinacions que es provaran, etc- , la tecla "p" per pausar temporalment la recerca (i "r" per restaurar-la) o la tecla "q" per sortir-ne definitivament.

b) ...utilitzant Hashcat i el mètode per diccionari amb les regles definides dins de l'arxiu "combinator.rule" que ve "de sèrie" amb el propi Hashcat. Observa pulsant la tecla "s" quantes combinacions haurà de provar el Hashcat fent servir aquestes regles i compara aquesta quantitat amb la que sortia a l'apartat anterior: ¿per què aquesta diferència? Si arribessis a esbrinar la contrasenya, elimina el potfile per poder començar de nou al següent exercici.

c) ...utilitzant Hashcat i el mètode per força bruta suposant que la contrasenya té una longitud entre sis i vuit caràcters i només conté lletres (majúscules o minúscules). Quan passi una estona, para de buscar perquè amb aquestes característiques no trobaràs mai la contrasenya.

d) ...utilitzant Hashcat i el mètode per força bruta sabent ara (perquè t'ho ha dit un "ocellet") que la contrasenya només té lletres minúscules i dígit i que té una longitud exacta de 8 caràcters. Si arribessis a esbrinar la contrasenya, elimina el potfile per poder començar de nou al següent exercici.

e) ...utilitzant Hashcat i el mètode per força bruta sabent que la contrasenya té exactament una longitud de 8 caràcters on les lletres minúscules ocupen exactament les posicions 1<sup>a</sup>,2<sup>a</sup>,3<sup>a</sup>,5<sup>a</sup>,6<sup>a</sup> i 7<sup>a</sup> i els dígit ocupen les posicions 4<sup>a</sup> i 8<sup>a</sup> . Si arribessis a esbrinar la contrasenya, elimina el potfile per poder començar de nou al següent exercici.

4.- Crea un arxiu de diccionari anomenat "tres.txt"que només tingui tres paraules: "taula", "cadira" i "porta" i crea un nou arxiu de regles anomenat "regles.txt" amb aquest contingut:

```
:  
l  
u  
$1  
$1 $2 $3  
^1^2^3  
c $1  
@a @e  
r [  
r ] ^x
```

a) Executa la comanda `hashcat tres.txt -r regles.txt -stdout` i observa què passa. ¿Té sentit el que veus?

b) Usa el diccionari i arxiu de regles anterior per intentar "crackejar" el següent hash (no hi ha salt de línia!:

```
$6$fQsjcwyB$OLOLnJ.9mgNuBBDxQTRu902qIgsWBupBfp/  
fn8.84XHv9mEcqxIvRXaXAhWBTP9SfzT87hSHMpACyb2ZmNMIM0
```

c) Digues quin seria el resultat d'aplicar cadascuna de les regles següents a la contrasenya "finestra":

```
c $2$0$2$0$!
c d $2$0$2$0$!
d c $2$0$2$0$!
c $2$0$2$0$! d
d $2$0$2$0$! c
$2$0$2$0$! d c
c ^3^2^1
```

**d)** Crea un altre diccionari anomenat "dos.txt" que tingui dues paraules: "cotxe" i "moto". Executa ara la comanda `hashcat -m 1800 -a 1 tres.txt dos.txt -j '^P $!' -k 'c $!' --stdout`. ¿Què veus per pantalla?

**NOTA:** És possible generar de forma pseudo-automàtica arxius de regles complexos a partir de màscares personalitzades gràcies a la comanda Maskprocessor ([https://hashcat.net/wiki/doku.php?id=rules\\_with\\_maskprocessor](https://hashcat.net/wiki/doku.php?id=rules_with_maskprocessor) ; <https://github.com/hashcat/maskprocessor>). Un exemple seria el següent:

```
mp -o arxiu.rule "$?a $?a"
mp -o arxiu.rule -1 ?s?d "$?a $?1 $?1"
```

Un únic atac amb diccionari usant l'arxiu "arxiu.rule" anterior seria equivalent a executar els dos següents atacs híbrids:

```
hashcat -m 1800 -a 6 hashes.txt dicc.txt $?a
hashcat -m 1800 -a 6 hashes.txt dicc.txt -1 ?s?d $?a?1?1
```

Però `mp` ens permet generar moltes més regles de les quals no hi ha equivalent amb atacs híbrids. Per exemple, ¿quines regles generaria la següent comanda: `mp -o arxiu.rule "?^?d $?a"? ¿O la comanda mp -o arxiu.rule "c d ^?d $?a" ?`

**5.-a)** Es poden provar varies màscares diferents, una darrera l'altra en una mateixa sessió de "crackeig", si aquestes s'escriuen en un fitxer "hashcat mask" i s'indica aquest fitxer com a darrer paràmetre en comptes de cap màscara explícita, així: `hashcat -m 1800 -a 3 /ruta/arxiuAmbHash.txt /ruta/arxiu.hcmask` Llegeix el següent apartat [https://hashcat.net/wiki/doku.php?id=mask\\_attack#hashcat\\_mask\\_files](https://hashcat.net/wiki/doku.php?id=mask_attack#hashcat_mask_files) per saber quin és el significat d'un hipotètic contingut d'un fitxer "hashcat mask" com el següent:

```
?u?d?d?d?d?s
?u?d,?d?u,??1?1?1?1?1?1
```

**b)** ¿Quin hauria de ser el contingut d'un arxiu "hashcat mask" tal que permetés generar combinacions de contrasenyes que tinguin entre 6 i 8 caràcters, el primer dels quals sabem que és una lletra majúscula, que l'últim és un dígit i que els caràcters d'entre mig poden ser lletres majúscules o minúscules?. Pista: es pot construir amb només tres línies.

**6.-a)** Utilitza el programa CUPP per generar un diccionari personalitzat segons les teves dades personals. Consulta <https://github.com/Mebus/cupp> per saber com fer-ho

**b)** Un altre generador de diccionaris és Statsprocessor (<https://github.com/hashcat/statsprocessor> ). A partir de la informació mostrada a la seva web i sabent el que són les cadenes Markov (una bona explicació es troba a <http://setosa.io/ev/markov-chains>, per exemple), dedueix quin és el criteri que fa servir aquesta comanda per generar les paraules del diccionari.

**c)** Un altre generador de diccionaris és Kwprocessor (<https://github.com/hashcat/kwprocessor> ). A partir del que s'explica a <https://cyberarms.wordpress.com/2018/02/13/creating-hashcat-keymap-walking-password-wordlists> , dedueix quin és el criteri que fa servir aquesta comanda per generar les paraules del diccionari

**d)** Un altre generador de diccionaris és l'script Python Wikigen (<https://github.com/zombiesam/wikigen> ) . A partir de la informació mostrada a la seva web, dedueix quin és el criteri que fa servir aquesta comanda per generar les paraules del diccionari.

**7.-a)** ¿ Per a què serveix l'aplicació Hashtopolis (<https://github.com/s3inlc/hashtopolis> )?

**aII)** ¿Quins components hardware té la màquina que venen a <https://terahash.com> ? ¿Quin software utilitza?

**b)** ¿Quina funció realitzen les diferents variants de la comanda *pwdump* llistades a <https://en.wikipedia.org/wiki/Pwdump> ?

**c)** Crea un full de càlcul amb el Libreoffice Calc, introdueix-hi qualsevol cosa a dins, dóna-li una contrasenya (això es pot fer al menú Eines → Protegeix el full de càlcul) i guarda'l amb format "Excel 97-2003". Utilitza ara l'script anomenat "office2hashcat.py", disponible a <https://github.com/stricture/hashstack-server-plugin-hashcat/tree/master/scrapers> , per tal d'extreure del full de càlcul a un fitxer de text el hash a endevinar pel Hashcat. Fes servir aquest fitxer de text com entrada del Hashcat per realitzar un atac amb diccionari (fent que la contrasenya vàlida estigui dins del diccionari emprat). ¿Funciona l'atac?

**8.-** En comptes d'haver d'endevinar una contrasenya a partir del seu "hash", una alternativa més fàcil és deixar funcionant un keylogger al sistema a espiar, alternativa que ens permetrà no només esbrinar la contrasenya d'inici de sessió sinó totes les demés (correu, xarxes socials, etc).

**a)** Segueix les instruccions d'instal·lació del keylogger "Logkeys" detallades a continuació (extretes de la seva web, <https://github.com/kernc/logkeys> i on s'ha suposat un sistema Fedora però en altres sistemes seria similar):

```
sudo dnf install automake make gcc gcc-c++
wget https://github.com/kernc/logkeys/archive/master.zip
unzip master.zip
cd logkeys-master
./autogen.sh
cd build
../configure
make
sudo make install
```

Descarrega't a més un fitxer que representa el mapa de teclat en espanyol simplement executant:  
wget [https://raw.githubusercontent.com/kernc/logkeys/master/keymaps/es\\_ES.map](https://raw.githubusercontent.com/kernc/logkeys/master/keymaps/es_ES.map)

**NOTA:** Per instal·lar logkeys només caldria fer: *cd logkeys-master/build && sudo make uninstall*

**NOTA:** Un altre keylogger, però aquest cas en forma de mòdul del kernel, es pot trobar a <https://github.com/jarun/keysniffer>

**b)** Executa en un terminal la comanda *sudo logkeys -s -m es\_ES.map --no-func-keys* . Seguidament, obre un altre terminal i executa *sudo tail -f /var/log/logkeys.log*. ¿Què veus?

**NOTA:** Per aturar el keylogger cal executar *logkeys -k* . Per veure el significat de més paràmetres, consulteu *man logkeys*

**NOTA:** Per aconseguir que el keylogger es posi en marxa automàticament en iniciar el sistema caldria construir un arxiu *.service* adient. Això ho estudiarem més endavant, en tractar el tema de Systemd