"CRACKING" DE CONTRASENYES

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.

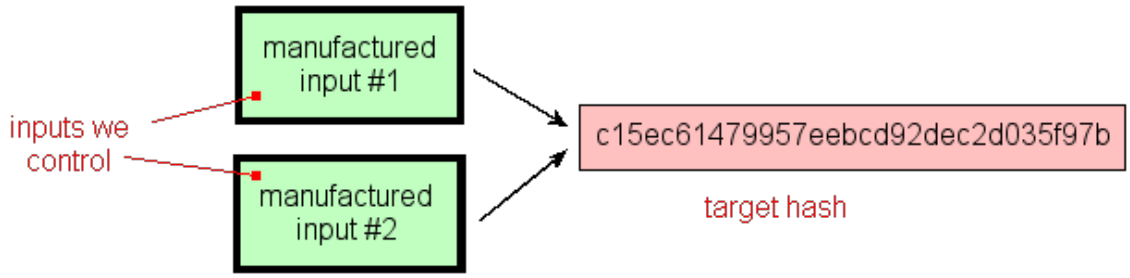
Colisiones

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 si: por pura estadística podría haber **colisiones:**

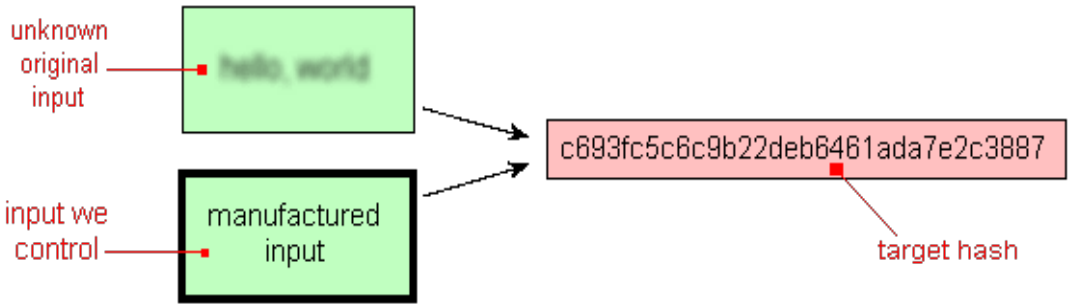
-El algoritmo MD5 genera un hash de 128 bits, por lo que hay "solo" 2^{128} hashes diferentes posibles
-El algoritmo SHA-1 genera un hash de 160 bits, por lo que hay "solo" 2^{160} hashes diferentes posibles
-El algoritmo SHA-258 genera un hash de 256 bits, por lo que hay "solo" 2^{256} hashes diferentes 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.

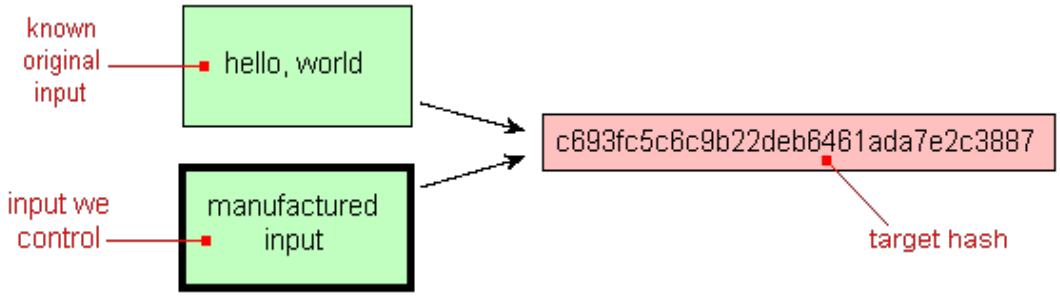
*_Collision Resistance_: Measures how difficult it is to pick two inputs that produce the same hash value. We manufacture both of the inputs in an attempt to coax the same hash value from each, and we don't care what the *particular* hash value generated is (just that they both match).



*_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 mediantes 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 "$" . Para generar un hash SHA-512 a partir de una combinación sal+contraseña dada que sea válido para ser almacenado en el archivo "/etc/shadow" (al tener el formato necesario *6valorsal$hashconjunto*) podemos ejecutar el siguiente comando:

openssl passwd -6 -salt valorsal 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","6salConcreta"))'* 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 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 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 podrem 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 fòrmula matemàtica que calcula la quantitat de variacions possibles donat un conjunt de caràcters concret (amb repeticions) en un número concret de posicions. Aquesta fòrmula és: $\mathbf{x^y}$, 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 fitxer 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/praetorian-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 (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
l o *u*	Passa a minúscula o majúscula totes les lletres de cada paraula del diccionari, respectivament
c	Posa en majúscula la primera lletra i en minúscula la resta de lletres de cada paraula del diccionari
C	Posa en minúscula la primera lletra i en majúscula la resta de lletres de cada paraula del diccionari
t	Posa a minúscula les lletres que estan a majúscula (i viceversa) de cada paraula del diccionari
TN	Posa a minúscula la lletra que ocupa la posició N si està a majúscula (o viceversa) de cada paraula
$X o *^X*	Afegeix "X" al final o al principi de cada paraula del diccionari, respectivament
sXY	Substitueix totes les ocurrències de "X" amb "Y" a cada paraula del diccionari
DN	Elimina la lletra que ocupa la posició N de cada paraula del diccionari
@X	Elimina "X" de cada paraula del diccionari
d	Genera, per cada paraula del diccionari, una paraula fruit de la seva duplicació ("hola"->"holahola")
pN	Afegeix N vegades darrera de cada paraula del diccionari ella mateixa
zN o ZN	Duplica la primera o darrera lletra de cada paraula del diccionari, respectivament
r	Inverteix cada paraula del diccionari
f	Afegeix al final de cada paraula del diccionari la seva inversió
{ o *}*	Mou una lletra de cada paraula del diccionari cap a l'esquerra o la dreta, respectivament
[o *]*	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 "màscares" 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?l?l?l 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?a es pot no haver de comprovar contrasenyes que tinguin exactament 5 caràcters sinò que es pot anar incrementant el número a provar pas a pas (primer contrasenyes amb un caracter 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 dígits numérics i les lletres a, b, c i d i un altre grup format per totes les lletres minúscules i els dígits 0,1,2 i 3, podríem afegir els paràmetres *-1=?dabcd* i *-2=?l0123* i llavors escriure com a màscara el següent, per exemple: *?1?1?2?1* .

NOTA: Tingueu en compte que a la definició del conjunt només estem indicant el tipus de caracter 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 máscara + 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

Hascat 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 línea 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 "crakejada". 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)