Algoritmos Hash e Integridad de archivos

Fernando Rodríguez Martín

Índice

1. Resumen y Debilidades de MD5 y SHA1

Las funciones hash son funciones denominadas como de ***un único sentido***, ya que se encargan de generar un ***hash o resumen*** a partir de una determinada entrada, y a partir del resumen es teóricamente imposible obtener la entrada de la función que lo ha generado.

Es importante destacar que, una función hash, **para una determinada entrada, siempre va a generar el mismo resultado**.

1.1. Algoritmos de generación de hashes

Se pueden utilizar diferentes algoritmos para generar este *resumen o hash*. Los distintos algoritmos van a determinar la longitud del hash y si pueden ocurrir ***colisiones*** para distintas entradas.

Una ***colisión*** se produce cuando, **para dos entradas diferentes de la función hash, esta devuelve el mismo hash**. Esto es un problema, ya que conociendo la forma en la que funciona el algoritmo, podemos hacer que dos entradas diferentes produzcan el mismo hash y así **falsear una prueba de integridad**.

Por tanto, hemos de usar algoritmos que no produzcan colisiones. A continuación, presento una lista de algoritmos típicos para generar hashes, y sus características:

Algoritmo	Tamaño salida	Colisiones
MD5	32 hex. char 128 bits	SI
SHA1	40 hex. char 160 bits	SI
SHA256	64 hex. char 256 bits	NO
SHA512	128 hex. char 512 bits	NO

Podemos ver, además, que todos los algoritmos, para cualquier entrada de un tamaño arbitrario, van a generar un hash **siempre de la misma longitud**, que depende del algoritmo dado.

1.2. Vulnerabilidades

Las dos principales vulnerabilidades que pueden presentar los algoritmos hash son las siguientes:

- **Ataques de fuerza bruta**: Poder encontrar la entrada que generó cierta salida de la función hash. Supone aplicar la función hash 2^n veces, donde n es la longitud del hash.
- **Colisiones**: Localizar dos entradas que den el mismo resultado de una función hash concreta. El mayor peligro es cuando el atacante puede generar un mensaje con sentido que colisione.

El problema con **MD5**, por tanto, es que presenta ambas vulnerabilidades. Lo primero, el resumen de MD5 es demasiado corto, por lo que computacionalmente es posible romper el hash por medio de un ataque de fuerza bruta.

Además, se descubrió que **MD5** presenta colisiones, por lo que, a partir de entonces, el algoritmo se considera vulnerable, y no debe usarse para problemas reales. Este artículo presenta como encontrar colisiones de MD5 con únicamente de 15 a 60 minutos de computación.

Por su parte, **SHA-1** no tiene el problema de la fuerza bruta, ya que un **ataque** contra SHA-1 por **fuerza bruta es impracticable** computacionalmente por el tiempo que puede tardar.

Sin embargo, en 2011, un grupo de investigación de Google sacó este paper donde explican como han comprometido **SHA-1** por la otra vía, **las colisiones**. Google liberó el código para poder generar dos pdf que produzcan el mismo hash, y desde entonces, no se debe usar este algoritmo.

1.3. MD5 y SHA1 para generar un hash - md5sum y sha1sum

Tanto los comandos de md5sum como sha1sum tienen las mismas opciones de entrada:

- **sha1sum/md5sum [OPTION] ... [FILE] ...**
- Si no añadimos un fichero de entrada, o si escribimos '-', va a leer de la entrada estándar. Cuando hayamos escrito la entrada, podemos hacer Ctrl + D para dejar de escribir.
- **-c, –check**: Calcula el hash para un fichero / entrada estándar, y comprueba si coincide con el hash introducido. Uso:
  - **$ sha1sum/md5sum -c fichero_con_hash <fichero_a_verificar**
  - Si no se añade la redirección, el comando por defecto lee de entrada estándar.
- Podemos especificar si queremos que el fichero se interprete como un fichero binario o como un fichero de texto con las opciones **-b, –binary** o **-t, –text**, respectivamente.

2. md5sum y sha1sum - Ejemplo práctico

Para demostrar que, efectivamente, md5sum y sha1sum, para la mayoría de casos, van a generar diferente hash para entradas distintas, vamos a probar a generar el hash con tres ficheros muy similares, para ver las diferencias. He creado los siguientes ficheros:

```
cat hola1.txt: 'holamundo'
cat hola2.txt: 'holaMundo'
cat hola3.txt: 'holamundo '
```

Para este ejemplo, vamos a utilizar los ficheros *hola1.txt* y *hola2.txt* con **MD5**, y los ficheros *hola1.txt* y *hola3.txt* con **SHA1**.

Por último comentar que estas pruebas se pueden recrear exactamente con las mismas cadenas, y los algoritmos devolverán **siempre el mismo resultado**. *(es lo que define a una función hash)*

2.1. md5sum - Demostración

He calculado el hash de los ficheros *hola1.txt* y *hola2.txt* con el algoritmo MD5, y he redirigido la salida de md5sum a los correspondientes ficheros ****.txt.md5***. Una vez hecho esto, comparo la salida con el comando **diff**, que muestra lo siguiente:

```
┌─[eskechi@pop-os]─(~/.../GSI/Practicas)
└─[19:15]-(T_T)-(53%)-[$] md5sum < hola1.txt > hola1.txt.md5

┌─[eskechi@pop-os]─(~/.../GSI/Practicas)
└─[19:18]-(^_^)-(51%)-[$] md5sum < hola2.txt > hola2.txt.md5

┌─[eskechi@pop-os]─(~/.../GSI/Practicas)
└─[19:18]-(^_^)-(51%)-[$] diff hola1.txt.md5 hola2.txt.md5
1c1
< f4d4c96eadd7e396578e055df1ea5e71  -
---
> 0cb0b5d4b19827e8ceaee23c1afda5ce  -
```

Por tanto, podemos ver que, pese a ser ficheros que solo se diferencian en una letra, los hashes que produce MD5 son completamente diferentes.

2.2. sha1sum - Demostración

Ahora, volvemos a hacer lo mismo pero esta vez con el algoritmo SHA-1, redirigiendo la salida de *sha1sum* a ***.txt.sha1***,y comparando los resultados con diff:

```
┌─[eskechi@pop-os]─(~/.../GSI/Practicas)
└─[19:20]-(^_^)-(51%)-[$] sha1sum < hola1.txt > hola1.txt.sha

┌─[eskechi@pop-os]─(~/.../GSI/Practicas)
└─[19:20]-(^_^)-(51%)-[$] sha1sum < hola3.txt > hola3.txt.sha

┌─[eskechi@pop-os]─(~/.../GSI/Practicas)
└─[19:20]-(^_^)-(51%)-[$] diff hola1.txt.sha1 hola3.txt.sha1
1c1
< 26b96d83f09a69e1d938176893b5e9ce16c436b3  -
---
> a8bf47cd6c03e8ff38c32a1b937cf69cf8617329  -
```

Como podemos ver, aunque estos ficheros se diferencian únicamente en un espacio al final, el hash calculado es radicalmente distinto.

3. Obtener hashcode de un fichero - MD5 y SHA1

Para este punto, he utilizado el fichero ***hola1.txt*** del apartado anterior, que contenía la cadena 'holamundo', y obtenemos la siguiente salida:

```
└─[19:53]-(^_^)-(39%)-[$] md5sum < hola1.txt
f4d4c96eadd7e396578e055df1ea5e71  -

┌─[eskechi@pop-os]─(~)
└─[19:53]-(^_^)-(39%)-[$] sha1sum < hola1.txt
26b96d83f09a69e1d938176893b5e9ce16c436b3  -
```

Podemos ver como coinciden con los del apartado anterior *(ya que es la misma entrada)*, y **MD5** produce una cadena de **32 caracteres**, mientras que **SHA-1 de 40**, ya que **la longitud viene determinada por el algoritmo**.

4. Obtener función hash - hashid

Ahora, vamos a utilizar los hashcode generados en el apartado anterior para intentar averigüar qué función hash ha generado dichos resumenes. ***hashid*** recibe como parámetro de entrada un ***hashcode***, y devuelve una lista de posibles algoritmos que lo hayan generado:

Figura 1: Resultado para el hash de 'holamundo' generado con SHA-1.

¿Como hace hashid para, a partir de un hash, averigüar el algoritmo? Ahora, es buen momento para recordar como, en el apartado anterior, comentabamos que cada algoritmo produce una cadena de una longitud determinada.

Pues bien, hashid **comprueba la longitud del hash** y busca **algoritmos** que generen un **hash con esa misma longitud**. Por eso, toda la lista de algoritmos que nos genera utilizan todos la misma longitud para el hash.

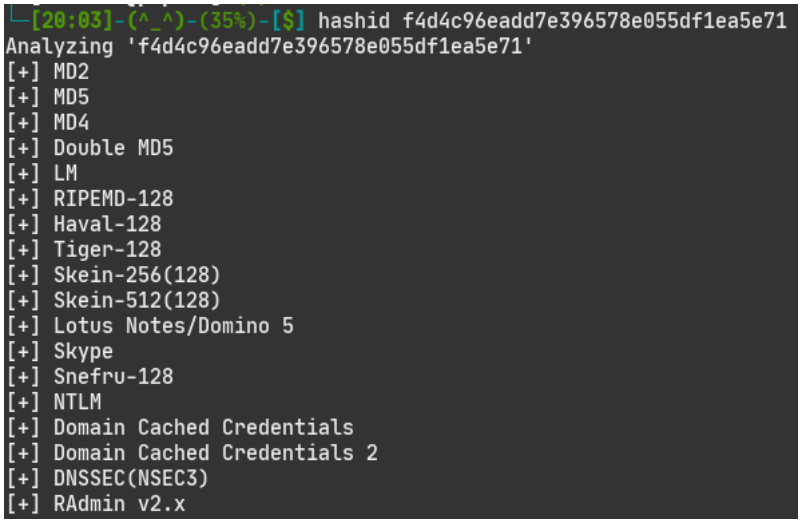


Figura 2: Resultado para el hash de 'holamundo' generado con MD5.

5. Hashcode y metadatos de un fichero

Ahora, vamos a ver si los metadatos de un fichero afectan al hash que pueda generar un algoritmo hash. Para ello, tenemos nuestro famoso fichero 'hola1.txt', que en principio tiene permisos **664** *(en hexadecimal)*, y vamos a generar su hash antes y después de modificar sus permisos a **777**.

Figura 3: Al modificar los permisos de usuario, el hash resultante ha cambiado.

5.1. Por qué es importante hacer un hash de los metadatos

Imaginemos que tenemos un archivo con un determinado hash, que fue generado por Mallet. Imaginemos también que ese archivo posee información ilegal, y Mallet acaba en un juicio por ella. Si Mallet pudiera modificar los metadatos, y no tuvieramos un hash de ellos, podría cambiar el creador a Alice, y que fuera ella quien acabara en prisión. Por tanto, **es importante hacer un hash de los metadatos**, para **verificar la integridad total del archivo**.

6. Fuerza bruta contra hashes - John The Ripper

Para este ejemplo, vamos a generar dos usuarios en nuestra máquina *mallet*, a *admin*, con contraseña *123456*, y a *root*, con contraseña *toor*.

Una vez que tenemos creados los dos usuarios con sus correspondientes contraseñas, podemos ver los hash de los usuarios en el fichero */etc/shadow*. Para ello, necesitamos tener permisos de superusuario. *(Ahora veremos por qué es peligroso que un usuario normal acceda a él...)*

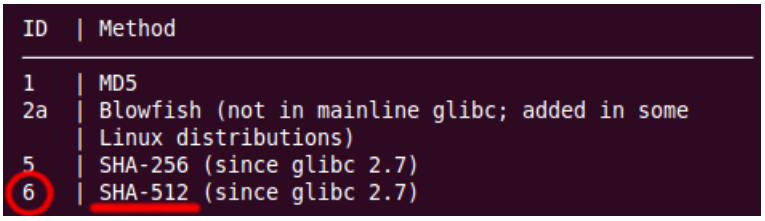
```
root:$6$0epr5E39$Q0NUTVwWznbGNG6IBoB0WSbjXpaUrMHFco5TCD/ntdoEgUvUNQ18H7dhPpR5RUk
90z.nW7xA.zbbY4XopnpOm1:19278:0:99999:7:::
```

```
admin:$6$BCNYHss8$Q7VQoyeOc2SVaBKAHVr6lLjW3wS0TYVf2WoU4h.VCERsudOR1RmNVV2DTBulwp
/YKOY3PB1dCBQnoiMBwPDDx.:19278:0:99999:7:::
```

Vemos que después de ambos nombres de usuario, la cadena siguiente siempre empieza por **6**. Si buscamos en la página del manual de ***crypt(3)***...

```
$id$salt$encrypted
```

Vemos que todas las contraseñas que se encripten van a tener el formato idsalt$encrypted. Por tanto, el **6** que vimos en el fichero ***shadow*** se tiene que corresponder con el ID del algoritmo utilizado sobre la contraseña.



De esta forma, averiguamos que el **algoritmo** de hashing **utilizado** sobre la contraseña **es SHA-512**. Sabiendo esto, podemos utilizar la herramienta ***John The Ripper*** para sacar la contraseña original.

6.1. John The Ripper

John The Ripper es una herramienta que realiza un ataque de **fuerza bruta basado en diccionarios** sobre un fichero de contraseñas encriptadas. Esta herramienta soporta diferentes algoritmos, en nuestro caso, nos interesa SHA-512. Para usar John The Ripper, simplemente **proporcionamos el archivo de contraseñas**:

```
mallet@mallet:~$ sudo john /etc/shadow
[sudo] password for mallet:
Loaded 3 password hashes with 3 different salts (generic crypt(3) [?/32])
mallet           (mallet)
toor             (root)
123456           (admin)
guesses: 3  time: 0:00:00:43 100.00% (2) (ETA: Thu Oct 13 21:30:55 2022)  c/s: 6
8.90  trying: 12345 - missy
```

Vemos que, efectivamente, la herramienta ha conseguido obtener el mensaje original. Esto se debe principalmente a que las contraseñas utilizadas eran muy débiles, por lo que hay que tener cuidado con qué contraseñas elegimos.

7. Automatización de hash

Para este apartado, vamos a crear un **programa python** que obtenga, de los ficheros cuyos nombres aparecen en el fichero argumento del programa, **el hash de su contenido y el hash de sus propiedades**. Cada fichero va a representar una línea del resultado, y los dos hashes van a estar separados por ;

```python
import os

f_in = open(input("Escriba el fichero de entrada: "), "r")
f_out = open(input("Escriba el fichero de salida: "), "w")
for line in f_in:
    line     = line.split("\n")[0]
    filehash = os.popen('sha1sum < '+line).read().split("\n")[0].split(" ")[0]
    prophash = os.popen('ls -l '+line+' | sha1sum').read().split("\n")[0].split(" ")[0]
    wrln     = line+': '+filehash+';'+prophash
    f_out.write(wrln+"\n")
    print(wrln)
```

Figura 4: El programa python.

Ahora, creamos un fichero que va a contener los nombres de los ficheros *'hola'* que utilizamos en la sección 2 de esta práctica. La salida para este fichero, por tanto, será:

```
┌─[eskechi@pop-os]─(~)
└─[22:20]-(^_^)-(99%)-[$] python3 buildhash.py
Escriba el fichero de entrada: input.txt
Escriba el fichero de salida: output.txt
hola1.txt: 26b96d83f09a69e1d938176893b5e9ce16c436b3;a23a26861f8a813494d34efb8e99ac2c92d83af3
hola2.txt: 3d0bae3b3e7c0c765d210e70c95dcf34f9c6f8e7;3cbbad6af46f21fa43f790e292dcda584c6fad96
hola3.txt: a8bf47cd6c03e8ff38c32a1b937cf69cf8617329;192563fc639c628c53809137e005763b550d8fb0
```

8. Comprobación de integridad automatizado

Ahora, creamos un programa que, para el fichero producido por la salida del anterior programa, compruebe para cada línea si se han producido cambios en el fichero o en sus propiedades.

```python
import os

f_in = open(input("Escriba el fichero de entrada: "), "r")
for line in f_in:
    file     = line.split(":")[0]
    filehash = line.split(":")[1].split(";")[0]
    prophash = line.split(":")[1].split(";")[1].split("\n")[0]
    os.popen('echo '+filehash+'  '+file+'> .filehash.sha1.tmp')
    os.popen('echo '+prophash+'  - > .prophash.sha1.tmp')
    filecmp  = os.popen('sha1sum -c .filehash.sha1.tmp').read().split("\n")[0]
    propcmp  = os.popen('ls -l '+file+' |sha1sum -c .prophash.sha1.tmp').read().split("\n")[0]
    print(file+': Fichero -> '+filecmp+'; Propiedades -> '+propcmp)
```

Figura 5: El programa python.

```
┌─[eskechi@pop-os]─(~)
└─[22:48]-(^_^)-(100%)-[$] python3 checkhash.py
Escriba el fichero de entrada: output.txt
hola1.txt: Fichero -> hola1.txt: La suma coincide; Propiedades -> -: La suma coincide
hola2.txt: Fichero -> hola2.txt: La suma coincide; Propiedades -> -: La suma coincide
hola3.txt: Fichero -> hola3.txt: La suma coincide; Propiedades -> -: La suma coincide
```

Figura 6: La salida del programa.