

FILE INTEGRITY

LEONARDO

TABLE OF CONTENTS

- Cosa intendiamo per integrità?
- Funzioni hash crittografiche
- Come si gestisce l'integrità di un file?
- Perché questo meccanismo funziona
- E nella pratica?

COSA INTENDIAMO PER INTEGRITÀ?

Nel contesto crittografico quando parliamo di **integrità** stiamo parlando di meccanismi che ci permettono di capire se un certo dato è stato cambiato oppure no.

Ci sono vari contesti in cui si effettuano dei controlli di integrità.

Tra questi troviamo anche i seguenti:

- rispetto ai messaggi di un protocollo di rete.
- rispetto ai file salvati in un dato file system.

In questo video ci concentreremo sul secondo contesto, ovvero quello dei file.

FUNZIONI HASH CRITTOGRAFICHE

I controlli di integrità utilizzano un particolare strumento ripreso dalla crittografia applicata: le **funzioni hash crittografiche**.

Ricordiamo che le **funzioni hash crittografiche** sono funzioni hash che mappano input di size arbitrario in output di size fissato con determinate proprietà.

$$H : \text{arbitrary size } X \longrightarrow \text{fixed size } Y = H(X)$$

Le proprietà specifiche delle funzioni hash crittografiche sono tre, e sono le seguenti:

- Preimage resistance
- Second preimage resistance
- Collision resistance

Preimage resistance (1/3)

Dato un valore di output y , è difficile trovare un valore di input x tale che

$$H(x) = y$$

Second preimage resistance (2/3)

Dato un valore di input x , è difficile trovare un altro valore di input \hat{x} tale che

$$H(x) = H(\hat{x})$$

Collision resistance (3/3)

È difficile trovare due input x_1, x_2 tali che

$$H(x_1) = H(x_2)$$

COME SI GESTISCE L'INTEGRITÀ DI UN FILE?

Supponiamo di avere un file f , e supponiamo di essere interessati a controllare l'integrità di tale file. A tale fine l'idea è quella di utilizzare una **funzione hash crittografica** H .

In particolare si procede nel seguente modo:

In particolare si procede nel seguente modo:

A partire dai byte del file f e dalla funzione H si calcola il valore di $H(f)$.

$$f \longrightarrow H(f) = f_{\text{check}}$$

In particolare si procede nel seguente modo:

A partire dai byte del file f e dalla funzione H si calcola il valore di $H(f)$.

$$f \longrightarrow H(f) = f_{\text{check}}$$

Dopo un po' di tempo per verificare se il file \hat{f} è uguale al file precedente f si calcola nuovamente il valore di $H(\hat{f})$

$$\hat{f} \longrightarrow H(\hat{f}) = \hat{f}_{\text{check}}$$

Se i risultati sono uguali, ovvero se

$$f_{\text{check}} = \hat{f}_{\text{check}}$$

allora possiamo dire che il file è integro, ovvero che nulla è cambiato nel periodo tra i due controlli effettuati.

Se invece i risultati non sono uguali, ovvero se

$$f_{\text{check}} \neq \hat{f}_{\text{check}}$$

allora possiamo dire che il contenuto del file è diverso da quello di prima.

Osservazione: Nel contesto del controllo di integrità il valore $H(f)$, ovvero il valore della funzione hash crittografica calcolata a partire dai byte di un file è chiamato il **checksum** del file.

PERCHÉ QUESTO MECCANISMO FUNZIONA

Questo modo di controllare l'integrità di un file riesce a funzionare per via delle peculiari proprietà possedute dalle funzioni hash crittografiche.

- **Preimage resistance**
- **Second preimage resistance**
- **Collision resistance**

Tra tutte queste proprietà, nel contesto del controllo dell'integrità dei file la proprietà che più ci interessa è la seconda, la **second preimage resistance**.

Dato un valore di input x , è difficile trovare un altro valore di input \hat{x} tale che

$$H(x) = H(\hat{x})$$

Essa ci dice che, dato il file f , il cui checksum è $H(f)$, sarà molto, molto difficile trovare un file \hat{f} diverso dal file f , il cui checksum è uguale a quello del file f .

$$H(f) = H(\hat{f})$$

Detto altrimenti, questa proprietà ci assicura che la funzione hash crittografica è un'ottima funzione per riassumere il contenuto dell'intero file in una quantità limitata di byte.

Il riassunto calcolato dalla funzione hash crittografica è tale che, cambiando anche solo un bit del file, il valore del checksum cambia drasticamente.

E NELLA PRATICA?

Vediamo adesso in pratica come funziona questo meccanismo di file integrity.

A tale fine possiamo creare un file chiamato `test.txt`, che contiene il seguente paragrafo, ripreso dal secondo libro di Dune, "Messia di Dune".

```
touch test.txt
```

```
echo "Non è certo al momento della loro creazione che gli  
Imperi mancano di uno scopo. Quando, invece, si sono fermamente  
consolidati, gli scopi si smarriscono e vengono sostituiti  
da vaghi rituali.
```

```
Dai detti di Muad'Dib, della Principessa Irulan." > test.txt
```

Essendo interessati a proteggere l'integrità di tale file, possiamo procedere con il calcolo della checksum.

Tale calcolo verrà effettuato sui singoli byte del file.

```
[leo@archlinux content]$ hexdump -C test.txt
```

00000000	4e 6f 6e 20 c3 a8 20 63	65 72 74 6f 20 61 6c 20	Non .. ce
00000010	6d 6f 6d 65 6e 74 6f 20	64 65 6c 6c 61 20 6c 6f	momento de
00000020	72 6f 20 63 72 65 61 7a	69 6f 6e 65 20 63 68 65	ro creazio
00000030	20 67 6c 69 0a 49 6d 70	65 72 69 20 6d 61 6e 63	gli.Impe
00000040	61 6e 6f 20 64 69 20 75	6e 6f 20 73 63 6f 70 6f	ano di unc
00000050	2e 20 51 75 61 6e 64 6f	2c 20 69 6e 76 65 63 65	. Quando,
00000060	2c 20 73 69 20 73 6f 6e	6f 20 66 65 72 6d 61 6d	, si sono
00000070	65 6e 74 65 0a 63 6f 6e	73 6f 6c 69 64 61 74 69	ente.conso
00000080	2c 20 67 6c 69 20 73 63	6f 70 69 20 73 69 20 73	, gli scop
00000090	6d 61 72 72 69 73 63 6f	6e 6f 20 65 20 76 65 6e	marriscon
000000a0	67 6f 6e 6f 20 73 6f 73	74 69 74 75 69 74 69 0a	gono sost
000000b0	64 61 20 76 61 67 68 69	20 72 69 74 75 61 6c 69	da vaghi
000000c0	2e 0a 0a 44 61 69 20 64	65 74 74 69 20 64 69 20	...Dai de
000000d0	4d 75 61 64 27 44 69 62	2c 20 64 65 6c 6c 61 20	Muad'Dib,
000000e0	50 72 69 6e 63 69 70 65	73 73 61 20 49 72 75 6c	Principes
000000f0	61 6e 2e 0a		an..
000000f4			

Per calcolare il checksum possiamo utilizzare la funzione hash crittografia **sha256** tramite il comando **sha256sum**

```
[leo@archlinux content]$ sha256sum test.txt
```

```
11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b92ff51  test
```

Come possiamo vedere, il risultato $H(f)$, il **checksum**,
è formato da 64 simboli.

11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b92ff51

Come possiamo vedere, il risultato $H(f)$, il **checksum**, è formato da 64 simboli.

11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b92ff51

Ogni due simboli formano un byte, ovvero 8 bit. In totale quindi abbiamo 32 byte, che sono

Come possiamo vedere, il risultato $H(f)$, il **checksum**, è formato da 64 simboli.

11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b92ff51

Ogni due simboli formano un byte, ovvero 8 bit. In totale quindi abbiamo 32 byte, che sono

$$32 \times 8 = 256 \text{ bit}$$

Come possiamo vedere, il risultato $H(f)$, il **checksum**, è formato da 64 simboli.

11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b92ff51

Ogni due simboli formano un byte, ovvero 8 bit. In totale quindi abbiamo 32 byte, che sono

$$32 \times 8 = 256 \text{ bit}$$

ed è per questo che il comando è chiamato **sha256sum**.

Per verificare l'integrità del file dobbiamo prendere l'output del comando precedente e salvarlo su un file, che chiamiamo **test.txt.sha256sum**, e poi utilizzare nuovamente il comando **sha256sum** ma questa volta con la flag -c

```
[leo@archlinux content]$ sha256sum test.txt > test.txt.sha256sum  
[leo@archlinux content]$ sha256sum -c test.txt.sha256sum  
test.txt: OK
```

Cosa succede adesso se cambiamo il valore anche di un solo byte del file originale? Ad esempio cosa succede se cambiamo la prima "N" in una "M"?

```
[leo@archlinux content]$ sed -i 's/^N/M/' test.txt
```


Se effettuiamo nuovamente il controllo otteniamo il seguente risultato

```
[leo@archlinux content]$ sha256sum -c test.txt.sha256sum  
test.txt: NON RIUSCITO  
sha256sum: ATTENZIONE: 1 codice di controllo calcolato NON corrisponde
```

Adesso il contenuto del file è il seguente

00000000	4d 6f 6e 20 c3 a8 20 63 65 72 74 6f 20 61 6c 20	Mon .. ce
00000010	6d 6f 6d 65 6e 74 6f 20 64 65 6c 6c 61 20 6c 6f	momento de
00000020	72 6f 20 63 72 65 61 7a 69 6f 6e 65 20 63 68 65	ro creazio
00000030	20 67 6c 69 0a 49 6d 70 65 72 69 20 6d 61 6e 63	gli.Impe
00000040	61 6e 6f 20 64 69 20 75 6e 6f 20 73 63 6f 70 6f	ano di unc
00000050	2e 20 51 75 61 6e 64 6f 2c 20 69 6e 76 65 63 65	. Quando,
00000060	2c 20 73 69 20 73 6f 6e 6f 20 66 65 72 6d 61 6d	, si sono
00000070	65 6e 74 65 0a 63 6f 6e 73 6f 6c 69 64 61 74 69	ente.conso
00000080	2c 20 67 6c 69 20 73 63 6f 70 69 20 73 69 20 73	, gli scop
00000090	6d 61 72 72 69 73 63 6f 6e 6f 20 65 20 76 65 6e	marriscon
000000a0	67 6f 6e 6f 20 73 6f 73 74 69 74 75 69 74 69 0a	gono sost
000000b0	64 61 20 76 61 67 68 69 20 72 69 74 75 61 6c 69	da vaghi
000000c0	2e 0a 0a 44 61 69 20 64 65 74 74 69 20 64 69 20	...Dai de
000000d0	4d 75 61 64 27 44 69 62 2c 20 64 65 6c 6c 61 20	Muad'Dib,
000000e0	50 72 69 6e 63 69 70 65 73 73 61 20 49 72 75 6c	Princess
000000f0	61 6e 2e 0a	an..
000000f4		

Notiamo che tra i due hexdump, l'unico byte che è cambiato è il primo, andando da **4e** (N) a **4d** (M).

Anche se è cambiato solo un byte, ricalcolando il checksum ottenuto con sha256 otteniamo una hash completamente diverso.

```
[leo@archlinux raw]$ sha256sum test.txt
```

```
2ad6bdcf205102236819f226607da1807fac53cc7210b2cf62b2176db1e61aff  test
```

Come possiamo vedere i due checksum sono
completamente diversi

(prima): 11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b92f-

(dopo) : 2ad6bdcf205102236819f226607da1807fac53cc7210b2cf62b2176db1e61a

Per riassumere, l'idea è la seguente:

Per riassumere, l'idea è la seguente:

Al tempo t si parte dal file **test.txt**, calcoliamo il checksum e lo salviamo da qualche parte

```
test.txt -> SHA256(contenuto di test.txt)  
         -> 11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b
```

Per riassumere, l'idea è la seguente:

Al tempo t si parte dal file **test.txt**, calcoliamo il checksum e lo salviamo da qualche parte

```
test.txt -> SHA256(contenuto di test.txt)  
          -> 11d71de73e9f163256d9d1957e6a7bf29ee12cfd7afc351723a2d55b4b
```

In un momento successivo $\hat{t} > t$, si ricalcola nuovamente il checksum dello stesso file, per vedere se il contenuto è cambiato

```
test.txt -> SHA256(contenuto di test.txt)  
          -> 2ad6bdcf205102236819f226607da1807fac53cc7210b2cf62b2176db1
```

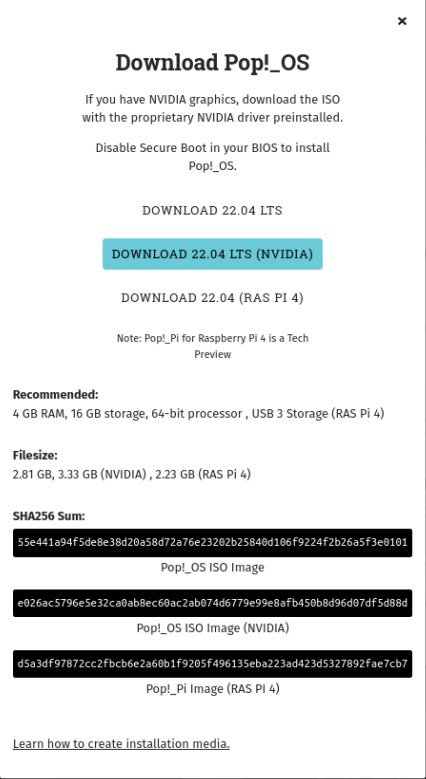

Per riassumere, l'idea è la seguente:

Alla fine si confrontano i due checksum calcolati, per vedere se sono uguali oppure no.

$$\text{checksum}_{\hat{t}} == \text{checksum}_t \quad ?$$

Se il checksum è uguale, allora vuol dire che il file **test.txt** non è cambiato in quel periodo di tempo, altrimenti vuol dire che è cambiato.

Ed è per questo che molto spesso assieme al download dei file troviamo anche esposti i checksum, per verificare se il download è andato bene.



The screenshot shows the 'Download Pop!_OS' page. It includes instructions for users with NVIDIA graphics to download the ISO with the proprietary NVIDIA driver preinstalled. It also mentions disabling Secure Boot in the BIOS. There are three download links: 'DOWNLOAD 22.04 LTS' (highlighted in blue), 'DOWNLOAD 22.04 LTS (NVIDIA)', and 'DOWNLOAD 22.04 (RAS PI 4)'. A note states that Pop!_Pi for Raspberry Pi 4 is a Tech Preview. The 'Recommended' section lists 4 GB RAM, 16 GB storage, 64-bit processor, and USB 3 Storage (RAS Pi 4). The 'Filesize' section lists 2.81 GB, 3.33 GB (NVIDIA), and 2.23 GB (RAS Pi 4). The 'SHA256 Sum:' section provides three checksums: for Pop!_OS ISO Image, Pop!_OS ISO Image (NVIDIA), and Pop!_Pi Image (RAS PI 4). At the bottom, there is a link to 'Learn how to create installation media.'

Download Pop!_OS

If you have NVIDIA graphics, download the ISO with the proprietary NVIDIA driver preinstalled.

Disable Secure Boot in your BIOS to install Pop!_OS.

DOWNLOAD 22.04 LTS

DOWNLOAD 22.04 LTS (NVIDIA)

DOWNLOAD 22.04 (RAS PI 4)

Note: Pop!_Pi for Raspberry Pi 4 is a Tech Preview

Recommended:
4 GB RAM, 16 GB storage, 64-bit processor , USB 3 Storage (RAS Pi 4)

Filesize:
2.81 GB, 3.33 GB (NVIDIA) , 2.23 GB (RAS Pi 4)

SHA256 Sum:

55e441a94f5de8e38d20a58d72a76e23202b25840d106f9224f2b26a5f3e0101
Pop!_OS ISO Image

e026ac5796e5e32ca0ab8ec60ac2ab074d6779e99e8afb450b8d96d07df5d88d
Pop!_OS ISO Image (NVIDIA)

45a3df97872cc2fbc6e2a60b1f9205f496135eba223ad423d5327892fae7cb7
Pop!_Pi Image (RAS PI 4)

[Learn how to create installation media.](#)

Pop!_OS download

Dato poi che un binario altro non è che un file, questo controllo può anche essere fatto rispetto ad un eseguibile, o a delle particolari sezioni di un eseguibile.

```
[leo@archlinux raw]$ file test
```

```
test: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),  
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,  
BuildID[sha1]=8f99e5e46456a976aee4795060a64cd7f521f64f, for GNU/Linux 4  
with debug_info, not stripped
```

```
[leo@archlinux raw]$ ./test
```

```
Hello World!
```

```
[leo@archlinux raw]$ hexdump -C test
```

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00 | .ELF.....
00000010  03 00 3e 00 01 00 00 00  40 10 00 00 00 00 00 00 | ..>.....@
00000020  40 00 00 00 00 00 00 00  08 47 00 00 00 00 00 00 | @.....C
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 25 00 24 00 | ....@.8..
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00 | .....@
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00 | @.....@
00000060  d8 02 00 00 00 00 00 00  d8 02 00 00 00 00 00 00 | .....
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00 | .....
00000080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00 | .....
00000090  18 03 00 00 00 00 00 00  1c 00 00 00 00 00 00 00 | .....
. . . .
```

```
[leo@archlinux raw]$ sha256sum test
```

```
b190b763758a5d8d6b4d9aaffffae55db7411df8a17d122adea1caf3b12ffe9f  test
```

Questi meccanismi di checksum sono alcuni dei meccanismi alla base della **protezione del software**.

Infine, notiamo come il controllo di integrità è legato alla scelta di una particolare funzione hash.

Dobbiamo quindi stare attenti se utilizziamo tale controllo per motivi di sicurezza, in quanto tipicamente col tempo le funzioni hash possono diventare vulnerabili, come è successo con la funzione hash **MD5**.

