Politecnico
di Milano

Mobilità e sicurezza delle reti: Autenticazione

Antonio Forzieri
Splunk Inc.

aforzieri@splunk.com

Autenticazione

- L'autenticazione è il servizio di sicurezza che permette di **garantire l'identità di qualcuno o qualcosa**.
- Si può voler autenticare:
  - **Un utente**: richiede **necessariamente** l'interazione da parte dell'utente:
    - Inserimento password, PIN, …
    - Scansione dell'impronta digitale, dell'iride, della voce …
  - **Una macchina**: non richiede alcuna interazione utente/macchina (es: autenticazione SSL).
  - **Dell'origine dei dati**: verifica che i dati ricevuti provengano da un origine (utente o macchina) **autentica** (es: e-mail firmate S/MIME o PGP).

Autenticazione: Sicurezza

- Si identificano **due requisiti di sicurezza** per i sistemi di autenticazione:
  - **Sicurezza in trasmissione**: durante il processo di autenticazione la credenziale non deve essere trasmessa in chiaro (plaintext), anzi non deve essere trasmessa affatto.
    - In caso contario un attaccante che riesca ad osservare il traffico in rete può **intercettarla** ed utilizzarla per **impersonare l'utente legittimo**.
  - **Sicurezza nel DB**: le credenziali non devono essere memorizzate in chiaro (plaintext) nel DB sul sistema deputato all'autenticazione
    - In caso contrario un attaccante che riesca a prendere possesso del sistema sul quale è presente il DB delle credenziali potrebbe impersonare **qualunque utente** la cui credenziale è li memorizzata.

Password MANTRA

"Password should be easy to remember and impossible to guess."

Real life: 2019

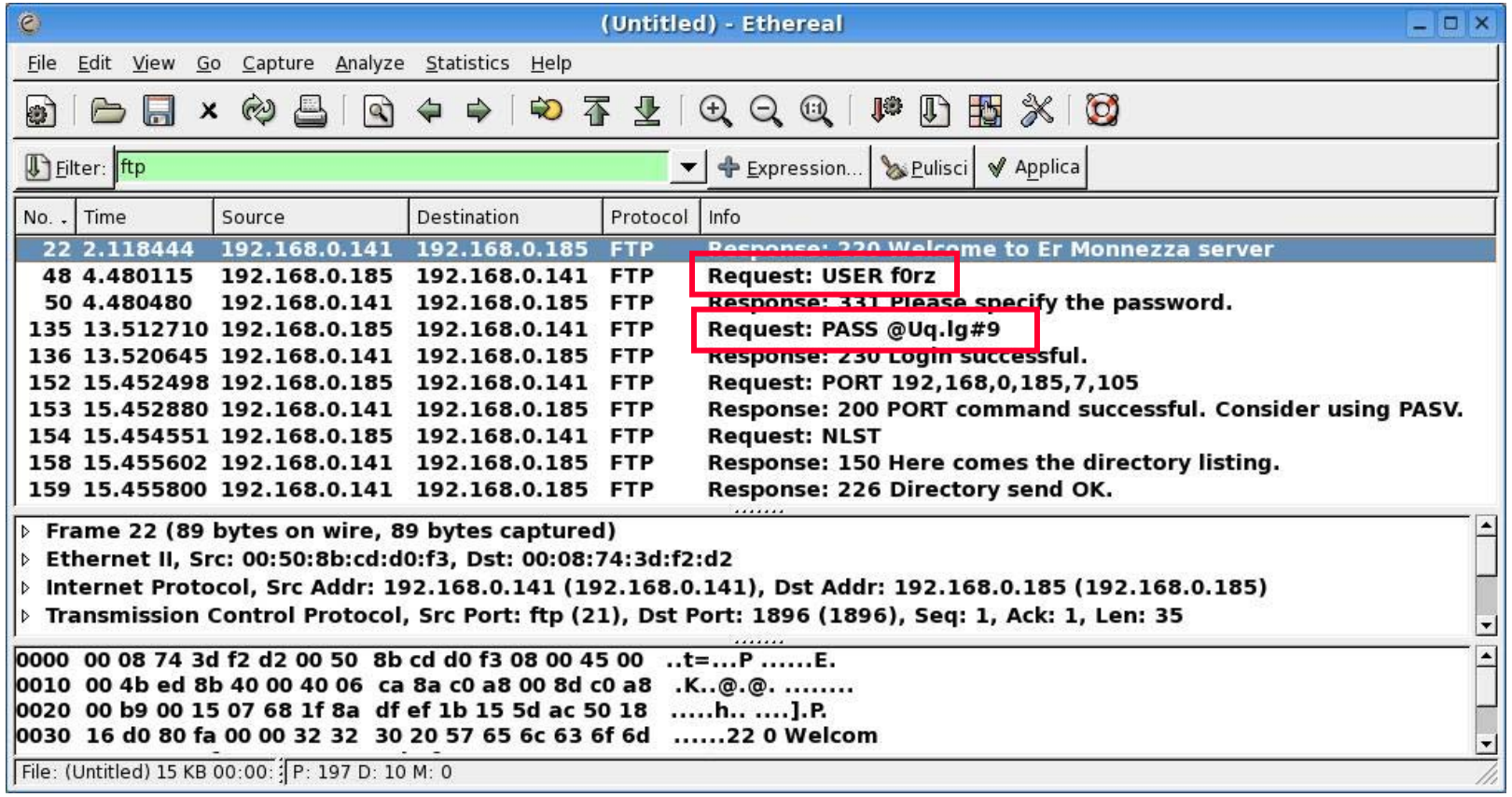
Sports, women's names, and food

The most popular passwords contain all the obvious and easy to guess number combinations (12345,111111,123321), popular female names (Nicole, Jessica, Hannah), and just strings of letters forming a horizontal or vertical line on a QWERTY keyboard (asdfghjkl, qazwsx, 1qaz2wsx, etc.). Surprisingly, the most obvious one — "password" — remains very popular; 830,846 people still use it.

Top 200 passwords from 2019 (click to expand)

	Password	Count
1.	12345	2812220
2.	123456	2485216
3.	123456789	1052268
4.	test1	993756
5.	password	830846
6.	12345678	512560
7.	zinch	483443
8.	g_czechout	372278
9.	asdf	359520
10.	qwerty	348762

Sicurezza password: sniffing



- In caso di autenticazione diretta o indiretta la password viene trasmessa in chiaro ed è facilmente **intercettabile**.

Sicurezza password: online guessing

- **Online guessing**: l'attaccante si collega a un determinato servizio (per esempio POP3 o FTP) e procede per tentativi al guessing (interazione diretta)
  - Contromisura:
    - Impedire che l'attaccante possa effettuare molti tentativi.
      - Un possibile approccio è quello di permettere *n* tentativi all'utente, dopodiché bloccare l'account per un certo periodo di tempo, aumentando il tempo di blocco dell'account per ogni insuccesso.
        - » Se dopo *n* tentativi l'account venisse bloccato fino all'intervento dell'amministratore di sistema, un attaccante potrebbe facilmente **bloccare tutti gli account**. (Denial of Service – DoS)
      - Il sistema potrebbe monitorare l'insuccesso nella procedura di autenticazione e lanciare un allarme (es: una mail all'amministratore).
      - Si può impostare un **ritardo** nel fornire la risposta all'utente in caso di accesso negato.

Sicurezza password: offline guessing

- Se l'attaccante riesce a ottenere una copia del file delle password può procedere al guessing **senza interagire con il sistema**:
  - Attacchi a forza bruta: anche ottimizzati (es: Xieve™ attack).
  - Attacchi a dizionario: si provano tutte le password contenute in un dizionario(o da leak più o meno pubblici) più alcune varianti (secondo politiche personalizzabili).
- Contromisure:
  - **Impedire che l'attaccante possa ottenere il file delle password**: per esempio permettendo la lettura di tale file solo all'amministratore.
  - **Rendere la funzione di hash lenta**: in questo modo si rende ogni singolo tentativo di guessing più lento
  - **Salting**: se il file delle password contiene l'hash della sola password un attaccante può attaccare tutte le password in parallelo. Una soluzione è quello di salvare <salt,h(salt,password)> in questo modo due password uguali avranno hash diverso.
  - **Pepper:** in aggiunta al salt, conservato in chiaro nel codice dell'applicazione.
  - **Cifrare il file delle password**: ma c'è il problema di dove memorizzare la chiave di cifratura.

Remember the SALT!

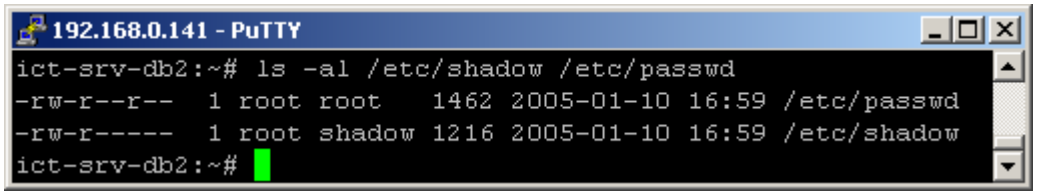
"Passwords are like chips better with SALT"

Sicurezza password: altri attacchi

- Gli attacchi visti non sono gli unici modi per ottenere la password di un utente legittimo ed impersonarlo. Sono possibili altri attacchi:
  - **Social engineer**: l'attaccante non tenta il guessing della password, ma la chiede all'utente.
  - **Shoulder surfing**: se l'attaccante è in prossimità di un utente può osservare la password digitata.
  - **Trojan horse**: può "mimare" l'interfaccia di login catturando la password.
  - **Key logger**:registrano tutto quello che viene digitato dall'utente, compresa la password (sia hardware che software).
  - **Van Eck sniffing**: le radiazioni emesse di un monitor, in teoria, sono visibili fino a 1 Km di distanza (TEMPEST)
  - **Mouse Pad survey**: molto spesso le password vengono scritte sul retro del mouse pad, dietro la tastiera o su un post-it attaccato al monitor.
  - **Keyboard Acoustic Emanations**: se l'attaccante riesce a registrare il suono emesso dalla tastiera durante la digitazione della password. Un recente studio mostra come l'80% delle password di 10 caratteri possa essere indivuato con solo 75 tentativi.

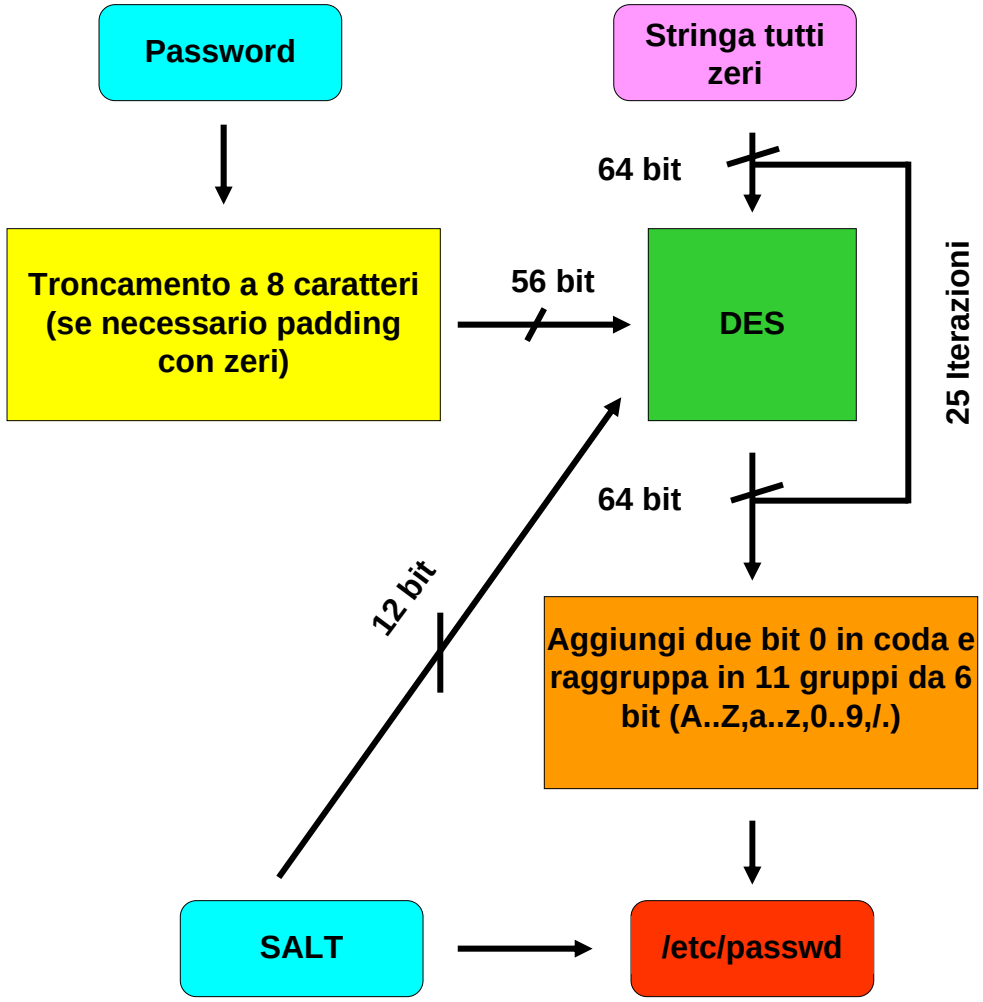
Password UNIX

- I sistemi UNIX memorizzano le informazioni relative all'account e le password in due file diversi:
  - Il file **`/etc/passwd`** contiene informazioni relative all'account utente ed è leggibile da tutti gli utenti
  - Il file **`/etc/shadow`** **(Linux)** o **`/etc/master.passwd`** **(*BSD)** contiene invece l'hash della password insieme ad altre informazioni ed è leggibile soltanto dall'utente **`root`**
- Oltre a rendere il file delle password inaccessibile agli utenti normali
  - La funzione di hash viene opportunamente rallentata;
  - Nel calcolo dell'hash si usa un salt ("hash salati");

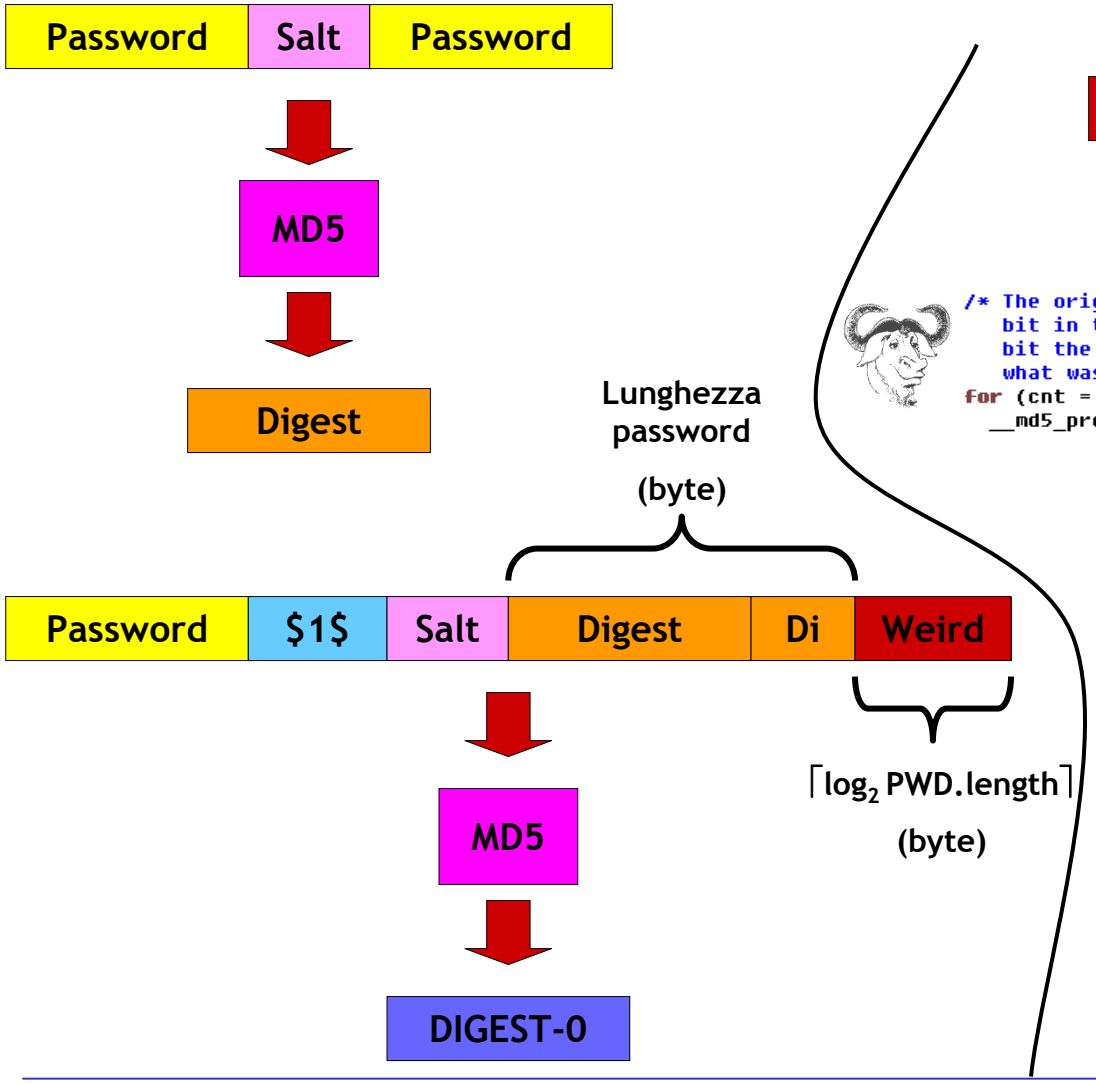


Password UNIX: crypt

- Per **offuscare** le password UNIX utilizzava la funzione **crypt** che utilizza una versione **modificata di DES**.
- Crypt esegue **25 cifrature DES modificate** per rallentare un eventuale attacco offline.
- Viene utilizzato un salt:
  - Password uguali hanno hash diversi. Rende più complessi attacchi a dizionario.
  - Il salt modifica la funzione di espansione del DES rendendo impossibile l'utilizzo di hardware dedicato (DES cracker, …).



Password Unix: crypt-MD5 (1)



Weird

↳Weird è costruito un byte alla volta. Ciascun byte è:

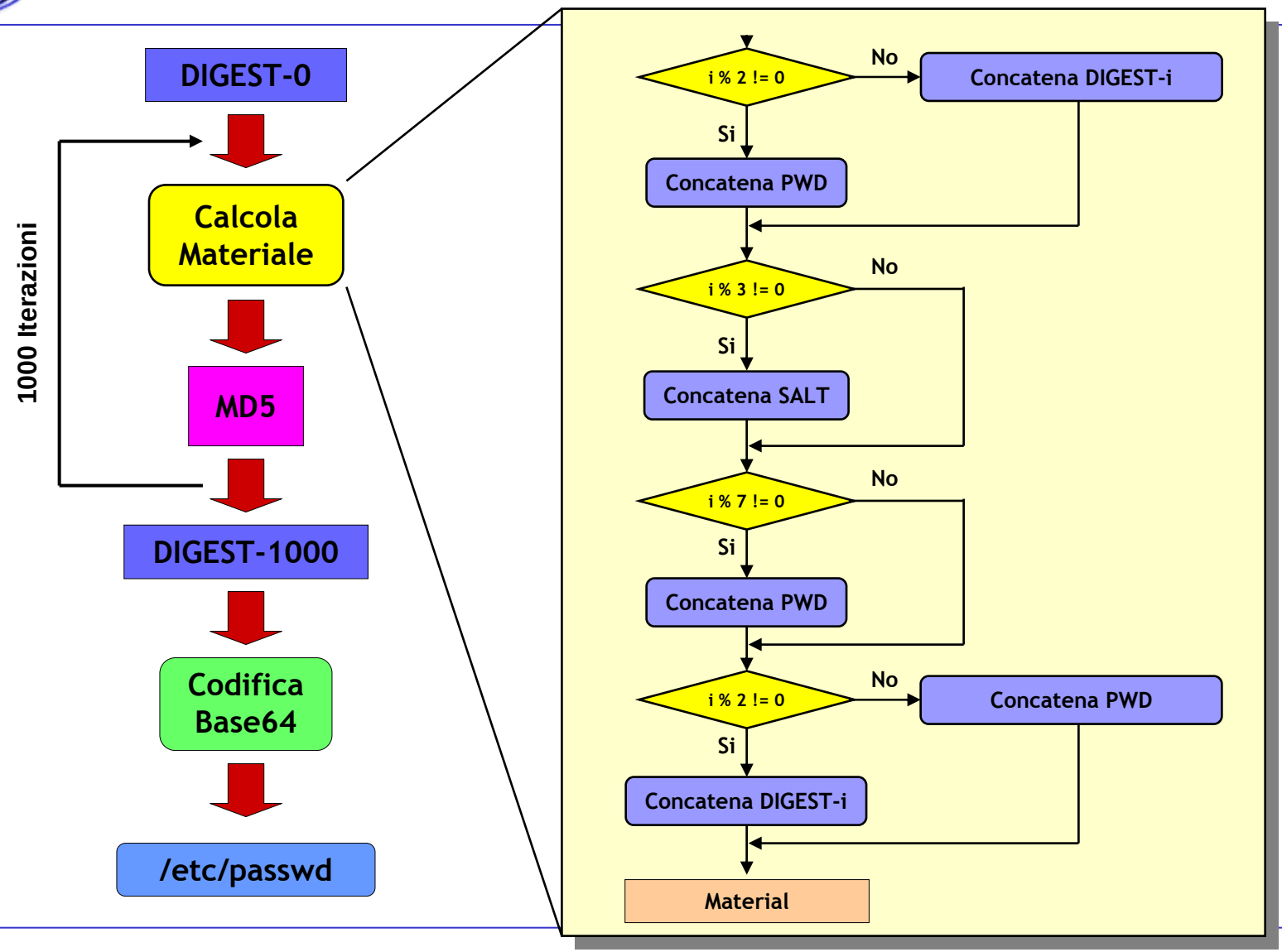
- ↳ Un byte nullo
- ↳ Il primo byte di PWD


```
/* The original implementation now does something weird: for every 1
   bit in the key the first 0 is added to the buffer, for every 0
   bit the first character of the key.  This does not seem to be
   what was intended but we have to follow this to be compatible.  */
for (cnt = key_len; cnt > 0; cnt >>= 1)
  __md5_process_bytes ((cnt & 1) != 0 ? (const char *) alt_result :
                       key, 1, &ctx);
```

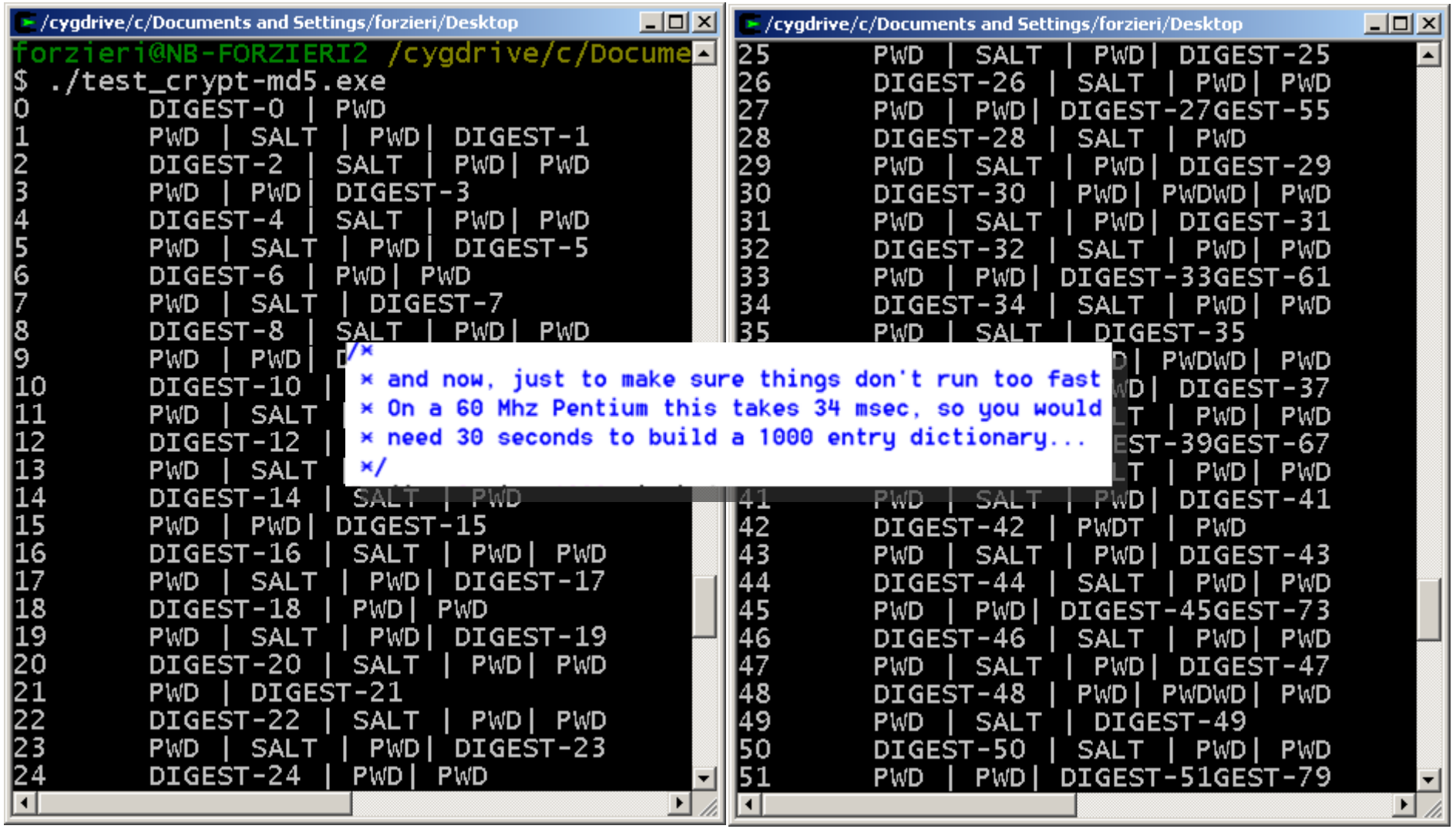
```
/* Don't leave anything around in vm they could use. */
memset(final, 0, sizeof(final));

/* Then something really weird... */
for (i = strlen(pw); i; i >>= 1)
        if(i & 1)
            MD5Update(&ctx, (const u_char *)final, 1);
        else
            MD5Update(&ctx, (const u_char *)pw, 1);
```

- Il **digest** della password è ottenuto mediante la funzione **crypt-MD5** un'evoluzione della funzione crypt (che si basa su DES).
- Il **salt** è al massimo 8 byte

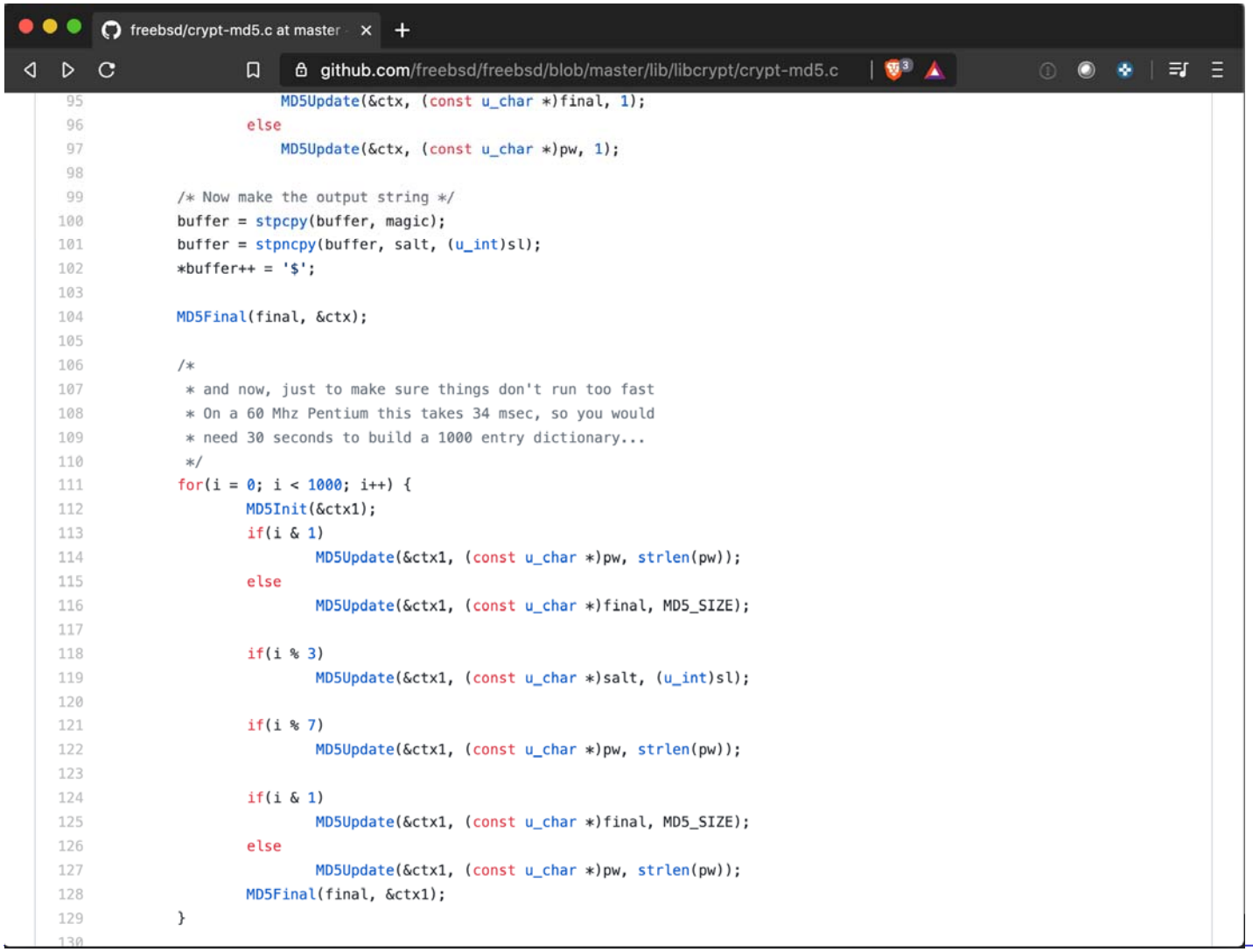
Password Unix: crypt-MD5 (2)



Password Unix: crypt-MD5 (3) (FreeBSD)



FreeBSD and BEERWARE license



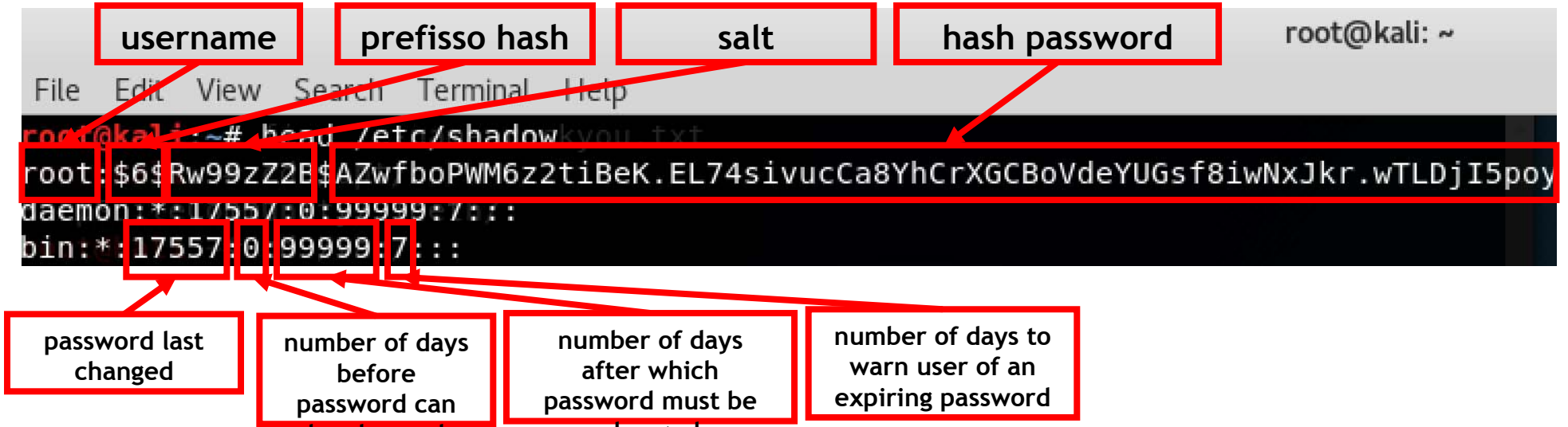

```
95                      MD5Update(&ctx, (const u_char *)final, 1);
96              else
97                      MD5Update(&ctx, (const u_char *)pw, 1);
98
99      /* Now make the output string */
100     buffer = stpcpy(buffer, magic);
101     buffer = stpncpy(buffer, salt, (u_int)sl);
102     *buffer++ = '$';
103
104     MD5Final(final, &ctx);
105
106     /*
107      * and now, just to make sure things don't run too fast
108      * On a 60 Mhz Pentium this takes 34 msec, so you would
109      * need 30 seconds to build a 1000 entry dictionary...
110      */
111     for(i = 0; i < 1000; i++) {
112             MD5Init(&ctx1);
113             if(i & 1)
114                     MD5Update(&ctx1, (const u_char *)pw, strlen(pw));
115             else
116                     MD5Update(&ctx1, (const u_char *)final, MD5_SIZE);
117
118             if(i % 3)
119                     MD5Update(&ctx1, (const u_char *)salt, (u_int)sl);
120
121             if(i % 7)
122                     MD5Update(&ctx1, (const u_char *)pw, strlen(pw));
123
124             if(i & 1)
125                     MD5Update(&ctx1, (const u_char *)final, MD5_SIZE);
126             else
127                     MD5Update(&ctx1, (const u_char *)pw, strlen(pw));
128             MD5Final(final, &ctx1);
129     }
130
```

Password UNIX: crypt-…

- La funzione **crypt-MD5** è sufficientemente robusta tuttavia:
  - La funzione di hash ha un tempo di esecuzione ben determinato
    - Con l'incremento delle potenze di calcolo il rallentamento potrebbe non essere più così efficace (legge di Moore);
  - Il salt è al massimo di soli 8 byte (collisioni di salt)
- Esistono altre varianti di crypt che permettono di impostare il "**rallentamento**" della funzione di hash:
  - **crypt-blowfish**: utilizza salt fino a 16 byte,ma ammette password lunghe al più **72 caratteri.** Il fattore di rallentamento (*cost*) è pari a $log_2 n$, con n numero di iterazioni di Eksblowfish (Expensive Key schedule Blowfish) - presente in alcuni *BSD.
  - **crypt-sha1**: utilizza salt fino a 64 bytes. Il fattore di rallentamento indica il numero di iterazioni hmac-sha1 eseguite – presente in alcuni *BSD.

Password UNIX - prefissi

- Nel file shadow troviamo gli hash delle password:



- Ogni prefisso hash identifica uno specifico hash:
  - 1: Crypt-MD5
  - 2: Bcrypt
  - $sha1$: crypt-Sha1
  - 4: sha1
  - 5: Crypt-SHA-256
  - 6: Crypt-SHA-512

Password UNIX – prefissi e python

- Potete utilizzare il modulo passlib di python su Linux per generare o verificare gli hash:

1. Python

```
M062761RDJGH6:test_hashes antonio_forzieri$ python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from passlib.hash import des_crypt
>>> hash = des_crypt.hash("toor")
>>> hash
'81kj8llaZQIRg'
>>> des_crypt.verify('toor','81kj8llaZQIRg')
True
>>> des_crypt.verify('wrong','81kj8llaZQIRg')
False
>>>
```

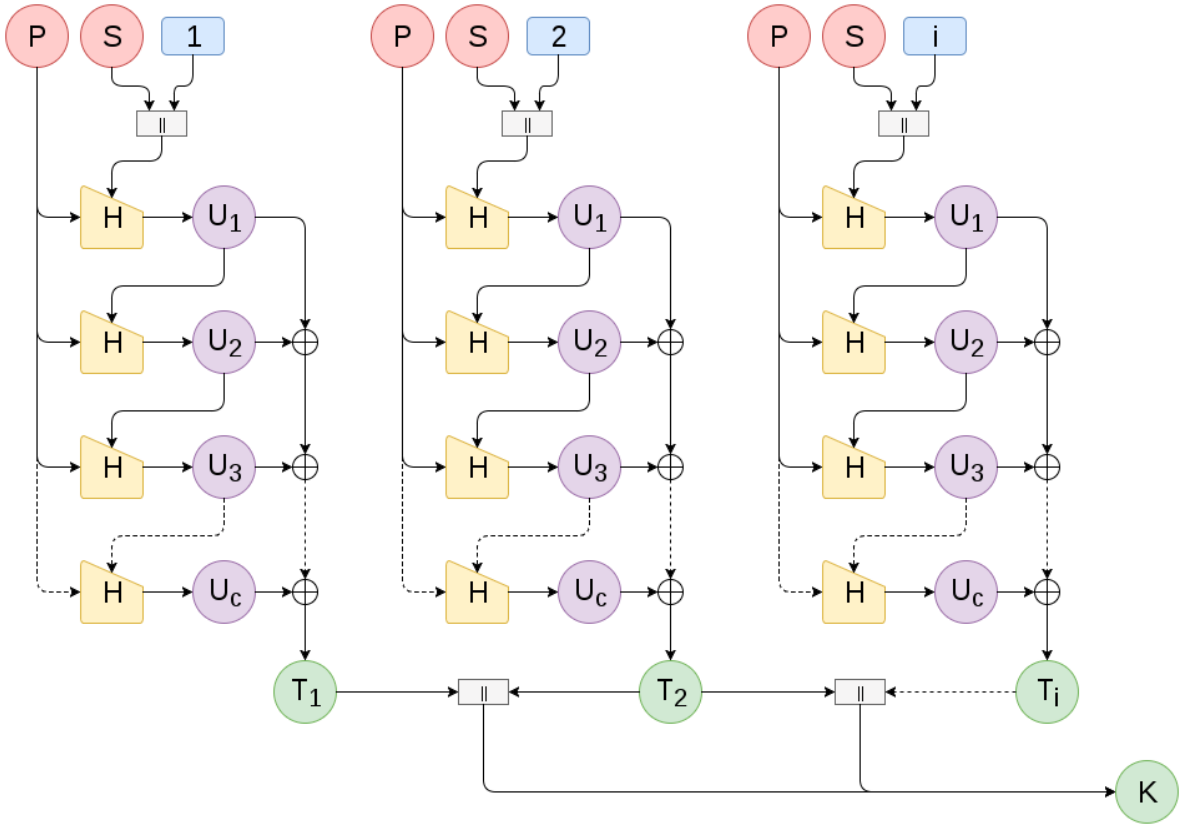
PBKDF2

- Password Based Key Derivation Function 2: è parte dello standard di RSA (**PKCS #5 v2.0**)
- Pubblicato anche da IETF come RFC2898
- Applica una Pseudorandom Function (e.g. HMAC-SHA1) a una **password**/**passphrase** congiuntamente a un **salt**
- Ripete il processo molte volte e produce una chiave crittografica in output.
- Utilizza **Key-Stretching**: insieme di tecniche per rendere una chiave debole (e.g. password o passphrase) una chiave debole più sicura, incrementando le risorse computazioni richieste per testare a bruteforce ogni possibile chiave.
- Semplice da implementare (e.g. 20 righe python)

PBKDF2

- P: password
- H: HMAC-*
- S: salt
- c: numero di interazioni

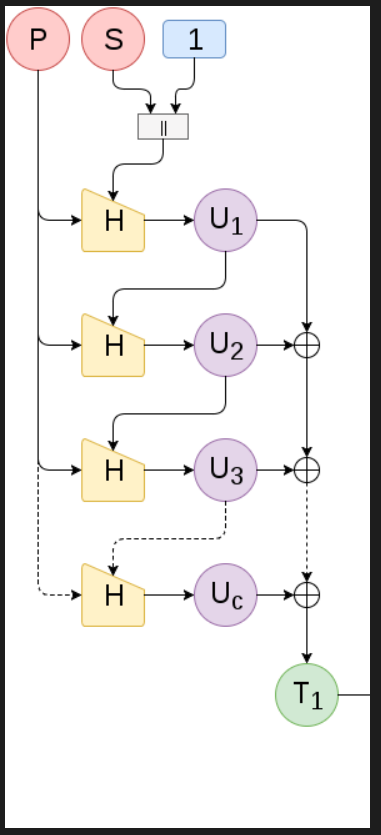
- MacOS 10.8+ usa di default:
  - H: HMAC-SHA512
  - c: 78740
  - S: 32 byte
  - K: 64 byte



PBKDF2: simple-pbkdf2

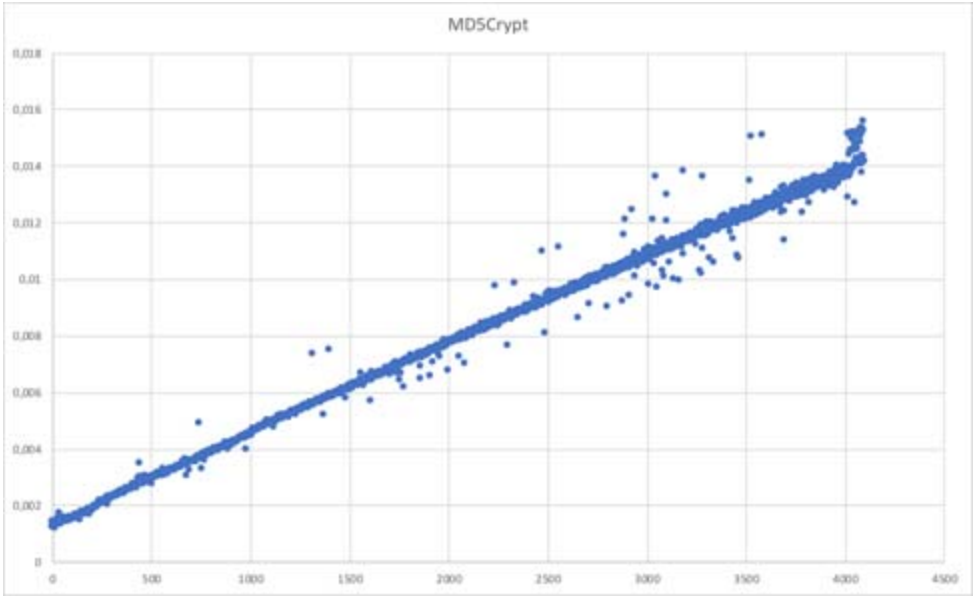
Users > antonio_forzieri > Desktop > pbkdf2.py > ...

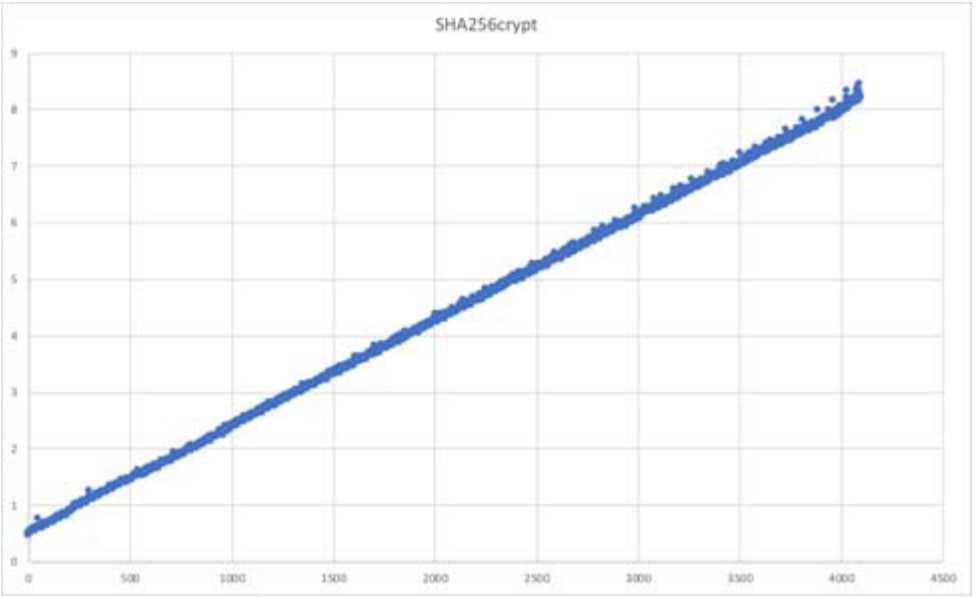
```python
import hmac
import hashlib
from struct import Struct
from operator import xor
from itertools import izip, starmap


_pack_int = Struct('>I').pack

def pbkdf2_hex(data, salt, iterations=1000, keylen=24, hashfunc=None):
    return pbkdf2_bin(data, salt, iterations, keylen, hashfunc).encode('hex')


def pbkdf2_bin(pwd, salt, iterations=1000, keylen=24, hashfunc=None):

    hashfunc = hashfunc or hashlib.sha1
    mac = hmac.new(pwd, msg=None, digestmod=hashfunc)
    def _pseudorandom(x, mac=mac):
        h = mac.copy()
        h.update(x)
        return map(ord, h.digest())
    buf = []
    for block in xrange(1, (keylen // mac.digest_size) + 1):
        rv = u = _pseudorandom(bytearray.fromhex(salt) + _pack_int(block))
        for i in xrange(iterations - 1):
            u = _pseudorandom(''.join(map(chr, u)))
            rv = starmap(xor, izip(rv, u))
        buf.extend(rv)
    return ''.join(map(chr, buf))[:keylen]

if __name__ == '__main__':
    print(pbkdf2_hex("mypassword","52fb16a4d9905df5a85c6c77e09e859ce9c435153c8f3f0d3e2995b454e681c4", iterations=78740, keylen=64, hashfunc=hashlib.sha512))
```



Password UNIX: un piccolo problema

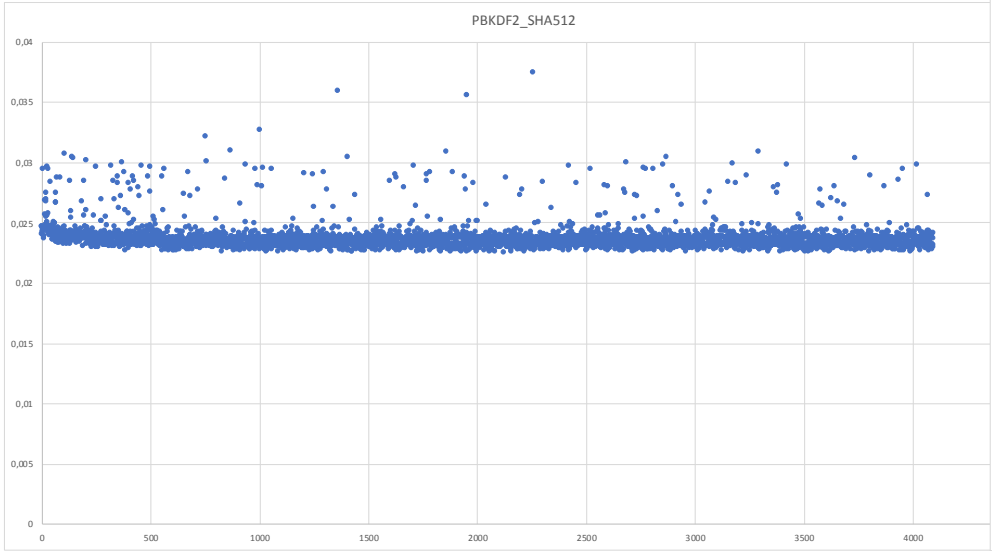
- Le funzioni di hash md5crypt, sha1, sha256crypt e sha512crypt hanno un tempo di esecuzione che dipende dalla lunghezza della password:
  - Password lunghe richiedono più tempo per essere processate

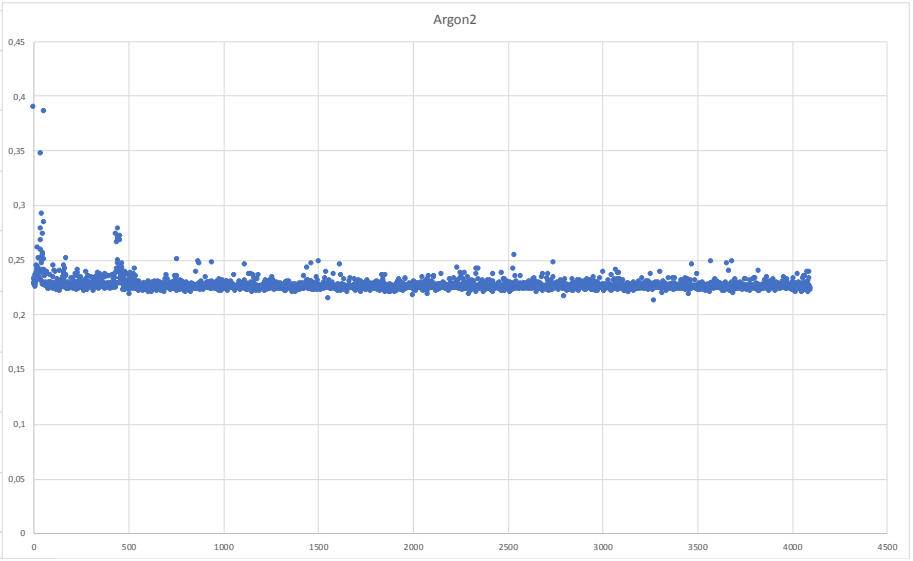




Password UNIX: un piccolo problema (2)

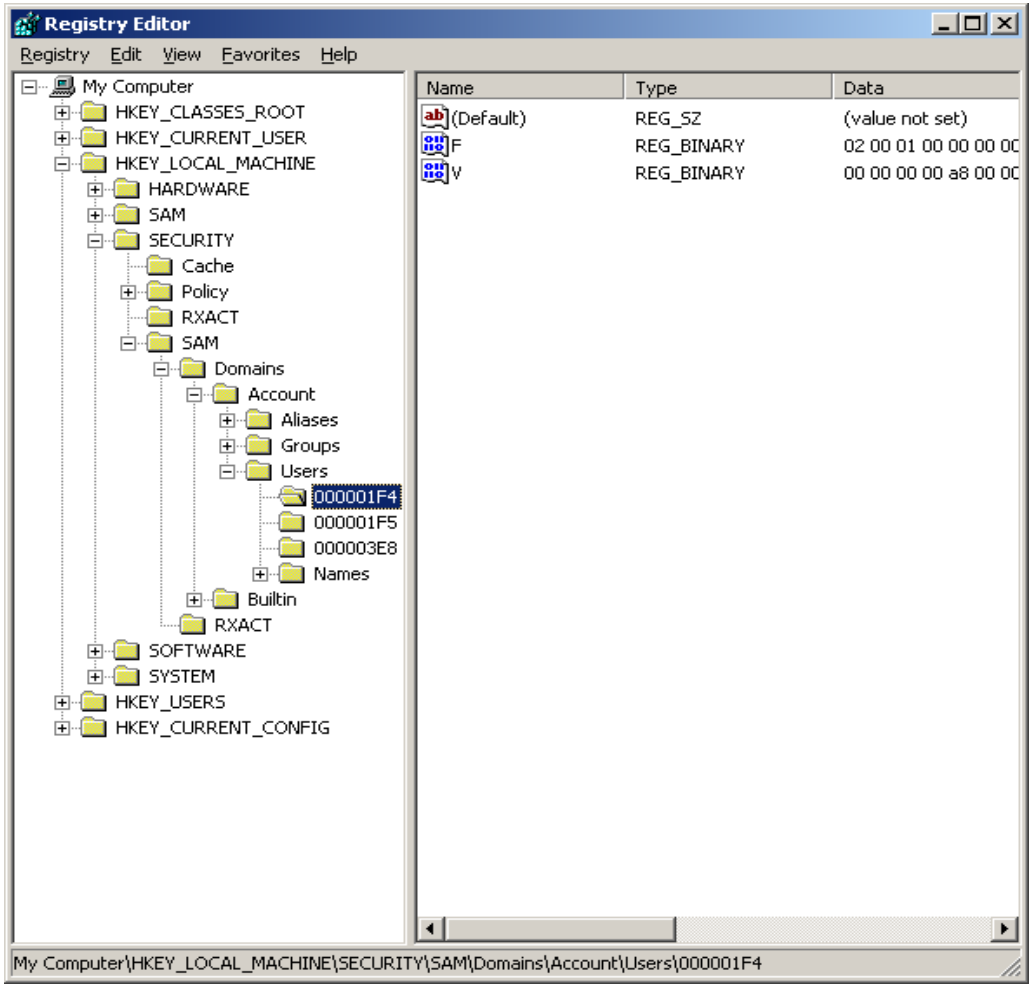
- Un tempo di esecuzione dipendente dalla lunghezza della password non introduce scenari di attacco significativi.
- É desidrabile tuttavia avere funzioni di hash con tempo di esecuzione non dipendente dalla password in input
- es: pbkdf2, argon2, scrypt





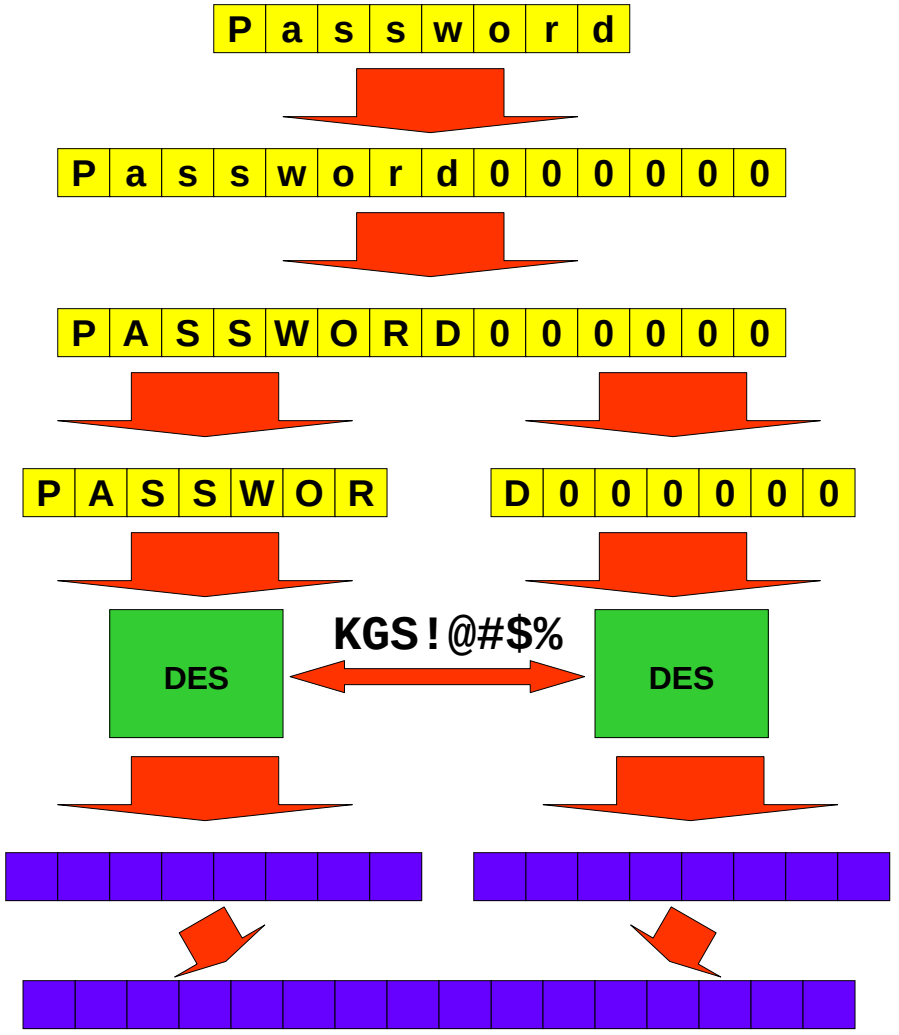
Password Windows

- Windows memorizza le informazioni sugli account utente all'interno del **System Account Manager** una parte del registro di Windows 2000/XP/2003.
  - Il **SAM** è conservato in HKLM\SECURITY\SAM ed è accessibile solo all'utente SYSTEM.
  - Una copia si trova nel file %windir%\system32\config\SAM
- Vengono memorizzati, in formato binario, **due tipi di hash:**
  - Hash Lanman
  - Hash NT Lanman



Hash Lanman

- L'hash Lanman della password è calcolato in questo modo:
  - la password viene troncata a 14 caratteri, oppure, se è più corta, sono inseriti in coda dei caratteri nulli;
  - i caratteri vengono **convertiti in maiuscolo**;
  - la stringa di 14 caratteri viene **divisa in due stringhe** da 7 caratteri;
  - ognuna delle due stringhe viene utilizzata come **chiave DES per cifrare una costante** di 64 bit (**0x4B47532140232425** corrispondente alla stringa **KGS!@#$%**);
  - i due risultati vengono **concatenati** in una stringa di 128 bit che costituisce l'hash.

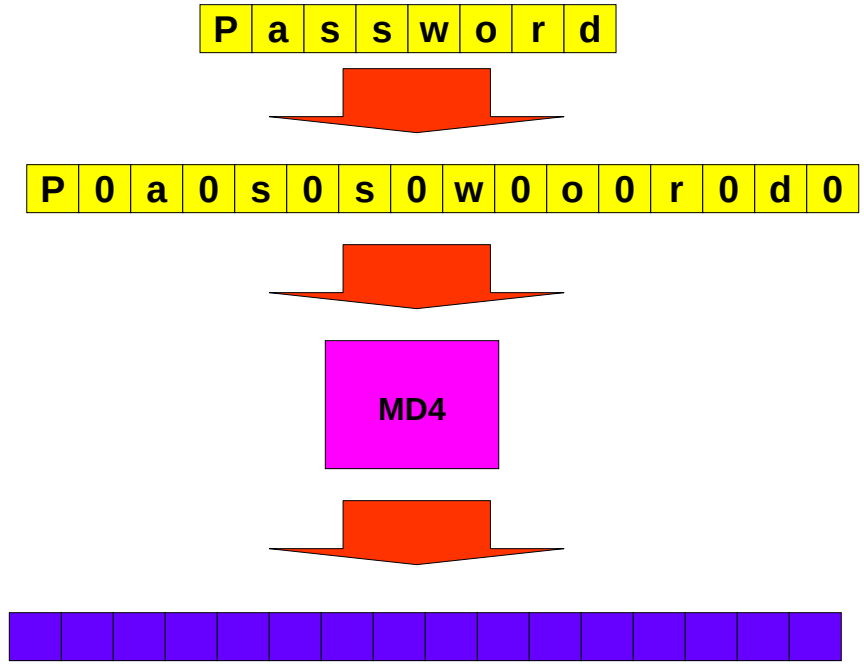


Hash Lanman:debolezze

- L'hash di 128 bit potrebbe sembrare complesso da attaccare. Ma:
  - Non viene usato un salt è possibile effettuare **attacchi a dizionario**
  - La conversione della password in lettere maiuscole fa **diminuire l'entropia massima accumulabile**. Con 95 caratteri stampabili si ha $\log_2 95 \approx 6{,}6$ bit per carattere. Convertendo in maiuscolo si perdono 26 possibili caratteri. Rimane pertanto accumulabile un'entropia di $\log_2 69 = 6{,}1$ bit per carattere
  - Dividendo la password in due metà:
    - **Si riduce lo spazio delle chiavi**. Con una password alfanumerica di 14 caratteri lo spazio delle chiavi sarebbe $36^{14} \approx 6{,}14 \times 10^{21} \approx$ **72,4 bit** con due password da 7 caratteri alfanumerici lo spazio si riduce a $2 \times 36^7 \approx 7{,}84 \times 10^{13} \approx$ **36,2 bit**!!!
    - Si può effettuare il cracking delle due metà **in modo indipendente**. Spesso la password è più corta di 14 caratteri, quindi la seconda metà è più facilmente individuabile.
    - Se la password è più corta di 8 caratteri la seconda metà dell'hash Lanman è un valore noto: **0xAAD3B435B51404EE**

Hash NT lanman (NTLM)

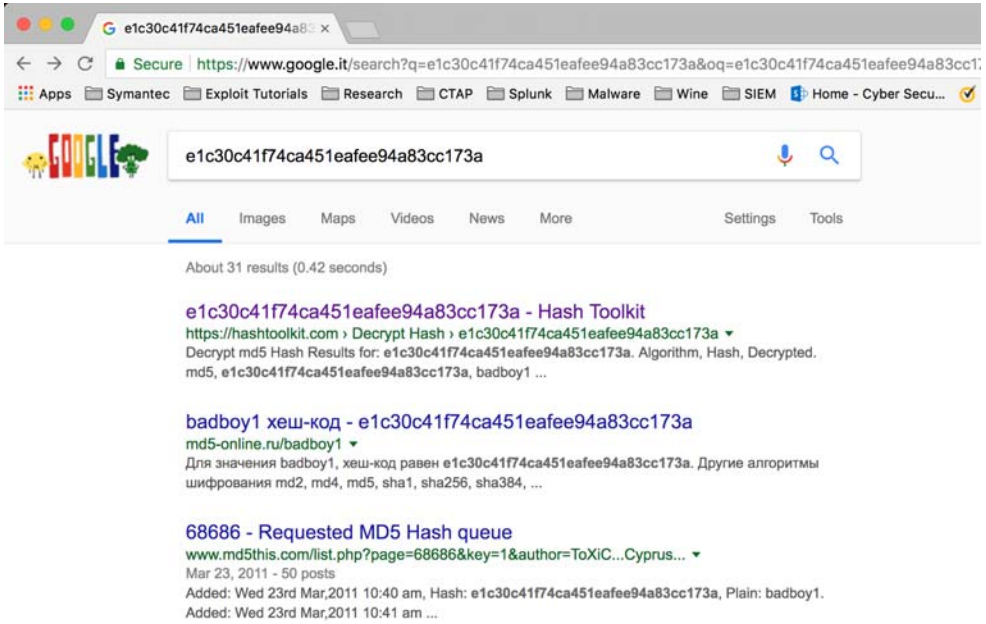
- L'hash NT Lanman della password è calcolato in questo modo:
  - La password in codifica ASCII viene **convertita in** codifica **Unicode** a 16 bit
    - La conversione da ASCII a Unicode si ottiene aggiungendo un byte nullo nel byte più alto.
  - Si effettua **l'hash MD4** della password in codifica Unicode.
  - Si hanno 128 bit (16 byte)



- L'hash NTLM sembra più robusto rispetto all'hash LM ma:
  - Non viene introdotto alcun salt: sono possibili attacchi a dizionario
  - Nel 1995 H. Dobbertin, Alf Swindles Ann hanno dimostrato come **sia possibile trovare** una collisione MD4 in **meno di un minuto** su un comune PC
  - Nel 1999 Dobertin ha dimostrato che le **prime due iterazioni di MD4 sono invertibili.**

Sicurezza password: ask Google FIRST!

- I motori di ricerca possono essere utilizzati per fare password cracking

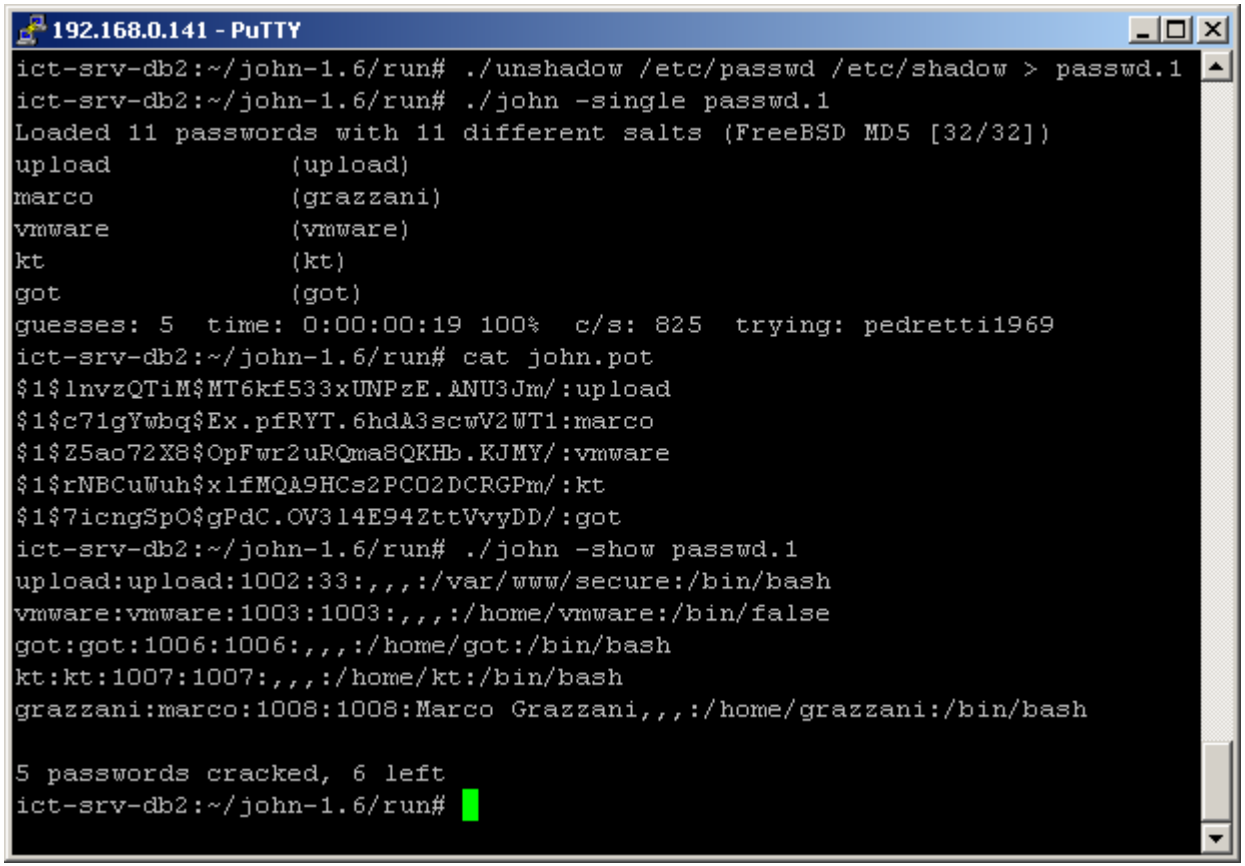


- Esistono script che automatizzano le stesse ricerche e fanno parsing HTML.

Sicurezza password: john the ripper (1)

- John the ripper è uno strumento per l'individuazione di password deboli:
  - Disponibile per diverse architetture:
    - 11 differenti Unix (Linux, FreeBSD, OpenBSD, Solaris, DIGITAL Unix, AIX, HP-UX, IRIX…);
    - DOS, Win32, BeOS, OpenVMS;
  - Effettua il cracking offline di:
    - password Unix (sono supportati diversi hash);
    - password kerberos AFS;
    - Password Windows NT/2000/XP/2003;
  - Diverse modalità di cracking:
    - **Single Crack Mode**: utilizza le informazioni di loging/GECOS;
    - **Wordlist Mode**: attacco basato su un dizionario (con regole di modifica password);
    - **Incremental Mode**: attacco a forza bruta configuarbile dall'utente;
    - **External Mode**: attacco scritto ad hoc dall'utente

Sicurezza password: john the ripper (2)



"5 password individuate utilizzando solo il campo login/GECOS in 19s"

Sicurezza password: john the ripper (3)

```
192.168.0.141 - PuTTY
ict-srv-db2:~/john-1.6/run# ./john -wordfile:/usr/share/dict/italian passwd.1
Loaded 6 passwords with 6 different salts (FreeBSD MD5 [32/32])
indovino         (test)
guesses: 1  time: 0:00:13:53 100%  c/s: 759  trying: zuzzurellone
ict-srv-db2:~/john-1.6/run# ./john -show passwd.1
upload:upload:1002:33:,,,:/var/www/secure:/bin/bash
vmware:vmware:1003:1003:,,,:/home/vmware:/bin/false
got:got:1006:1006:,,,:/home/got:/bin/bash
kt:kt:1007:1007:,,,:/home/kt:/bin/bash
grazzani:marco:1008:1008:Marco Grazzani,,,:/home/grazzani:/bin/bash
test:indovino:1009:1009:,,,:/home/test:/bin/bash

6 passwords cracked, 5 left
ict-srv-db2:~/john-1.6/run# cat john.pot
$1$lnvzQTiM$MT6kf533xUNPzE.ANU3Jm/:upload
$1$c71gYwbq$Ex.pfRYT.6hdA3scwV2WT1:marco
$1$Z5ao72X8$OpFwr2uRQma8QKHb.KJMY/:vmware
$1$rNBCuWuh$xlfMQA9HCs2PCO2DCRGPm/:kt
$1$7icngSpO$gPdC.OV3l4E94ZttVvyDD/:got
$1$FIfdZ1Wc$2HYHNoSdHupbo7ki/2ouX/:indovino
ict-srv-db2:~/john-1.6/run#
```

"Più del 50% delle password individuate in meno di 15 minuti"

"un'altra password individuata utilizzando un dizionario italiano con 116.878 termini"

Sicurezza password: Hashcat

- Strumento gratuito ed Opensource (licenza MIT)
- Multi OS (Linux, Windows MacOs)
- Multi Piattaforma: CPU, GPU, FPGA e qualunque cosa sia supportato da OpenCL
- Multi Hash: può fare password guessing su milioni di hash contemporaneamente
- Multi Device: supporta device diversi sullo stesso host (e.g. CPUs + GPUs)
- Supporta Guessing Distribuito
- 200+ tipologie di hash supportate

Sicurezza password: Hashcat (2)

- Modalità supportate:
  - Attacco a Dizionario
  - Combinator Attack
  - Brute Force
  - Mask Attack
  - Hybrid Attack
- Attacco a dizionario: viene fornito un dizionario e il tool calcola l'hash di ciascuna password nel dizionario per verificare se questo sia uguale a uno degli hash da indovinare.
  - Pro: Molto Veloce
  - Contro: l'efficacia dipende dalla bontà del dizioanario

Sicurezza password: Hashcat (3)

- **Combinator Attack**: vengono forniti due dizionari (ed eventualmente regole da applicare), il tool crea tutte le possibili combinazioni di coppie di password e ne effettua l'hash per verificare se questi siano uguali all'hash da indovinare:
  - Pro: molto efficace e veloce.
  - Contro: l'efficacia dipende dalla bontà dei dizionari.
  - Es: ./hashcat -m 0 -a 1 hash.txt dict1.txt dict2.txt
- **Bruteforce**: tenta tutte le combinazioni per un dato spazio delle password.
  - **Pro**: attacco più semplice in assoluto.
  - **Contro**: Poco adatto in caso di password lunghe.

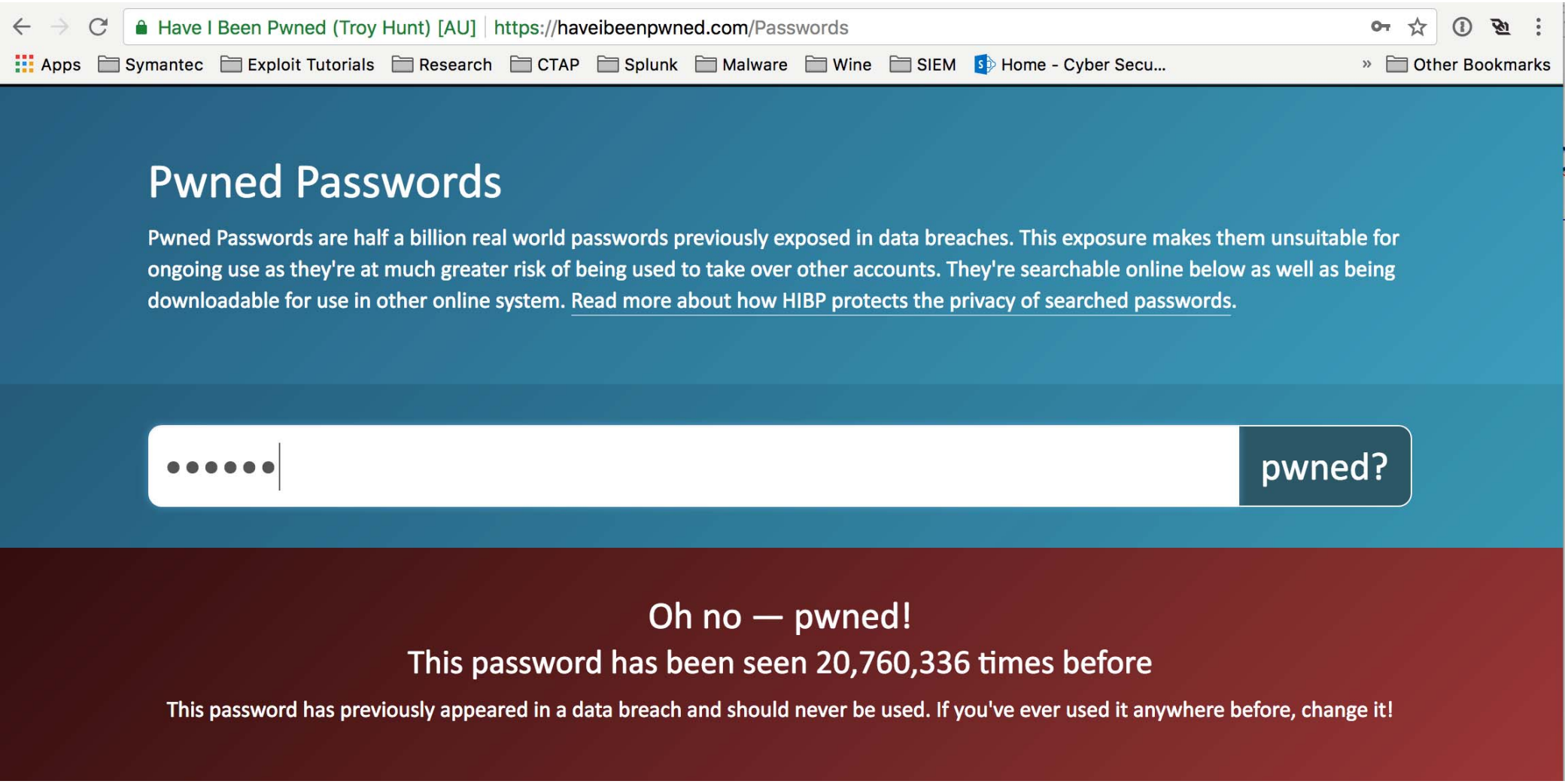
Sicurezza password: Hashcat (4)

- Mask Attack:
  - Supponiamo di dover indovinare una password di lunghezza 9 (solo lettere e numeri). Lo spazio delle chiavi ha dimensioni 62^9.
  - Utilizzando hardware che ci permette di provare 100Mhash/s richiederebbe circa 4 anni.
  - Ma di solito le persone utilizzano formati specifici: es. 5 lettere e 4 numeri (es: Julia1984).
  - Questo riduce lo spazio delle chiavi a 52*26*26*26*26*10*10*10*10 e richiederebbe solo circa 40' a 100Mhash/s
  - **Pro**: molto efficiente se si conosce lo schema della password
  - **Contro**: funziona solo per un set limitato di password.

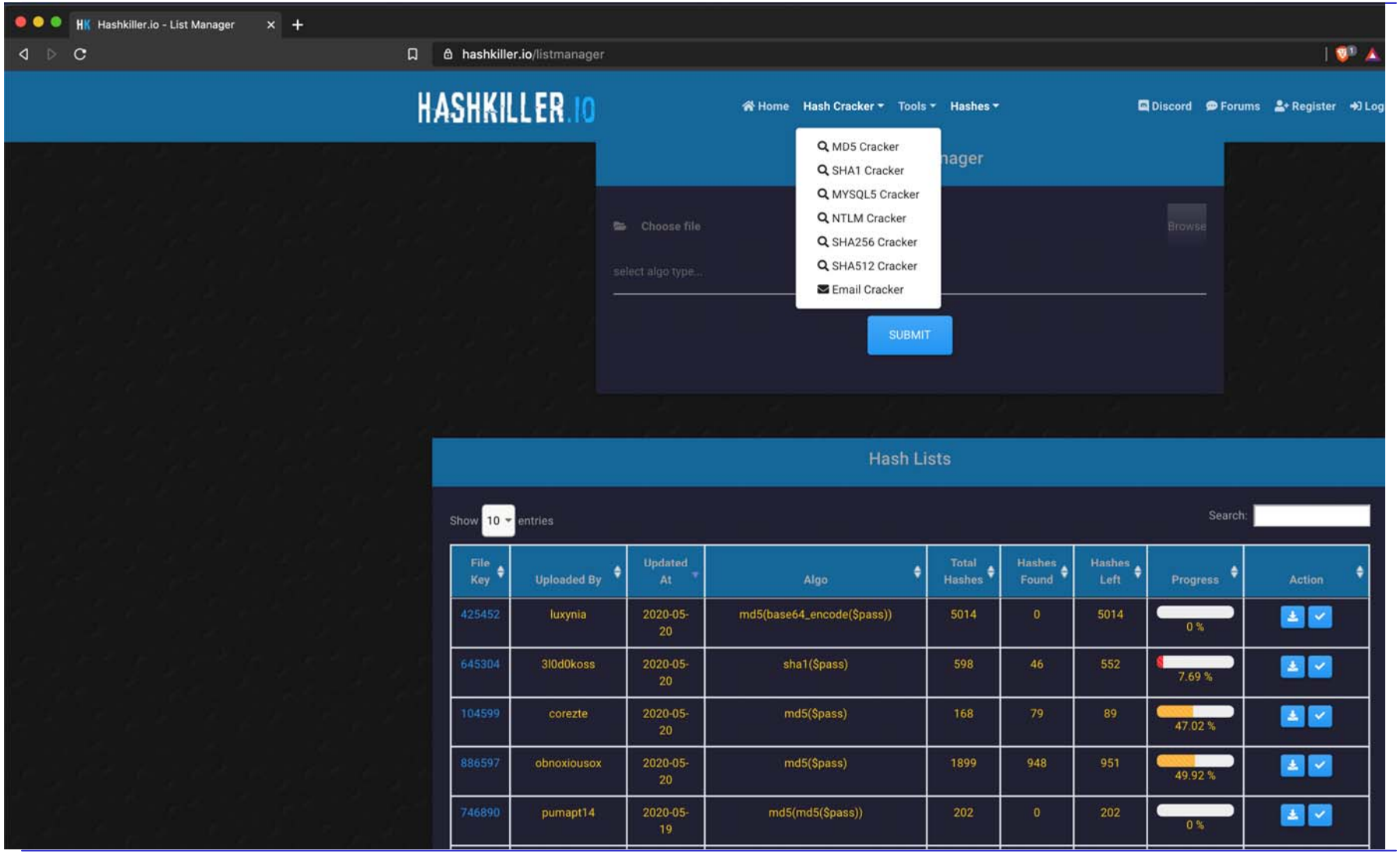
Sicurezza password: Hashcat (5)

- Hybrid Attack:
  - È essenzialmente un attacco Combinator:
    - Una metà della password viene presa da un dizionario
    - L'altra metà da un attacco a forza bruta
    - Es: hashcat hash.txt -a 6 example.dict ?d?d?d?d
      - Se example.dict contiene "password, hello"
      - Genera: password0000, password0001, password0002…
    - Es2: hashcat hash.txt -a 7 ?d?d?d?d example.dict
      - 0000password
      - 0001password
      - 0002password
      - …

HaveIbeenPwned by TroyHunt



Hashkiller (https://hashkiller.io)



Argon2: the new standard

- Vincitrice della **Password Hashing Competition** (2013-2015).
- É una key derivation function che permette di personalizzare:
  - Il tempo di esecuzione (**t -> iterations**)
  - La memoria richiesta (**m -> memorySizeKB**)
  - Il livello di parallelismo (**p -> parallelism**)
- Le funzioni viste fino ad adesso (e.g. PBKDF2) permettono di rallentare l'esecuzione della funzione ma NON di rendere l'esecuzione più o meno costosa sotto il profilo della memoria richiesta.
- GPGPU o ASIC permettono di avere una grande efficienza di calcolo parallelo.

Argon2: inputs

- Prende in input i seguenti parametri:
  - Password: **P** [0-2^{32}-1 bytes]
  - Salt: **S** [8-2^{32}-1 bytes -> 16 bytes raccomandati]
  - Parallelism: **p** [1-2^{24}-1 computational chains/threads]
  - Tag Length: **τ** [4 - 2^{32} – 1 bytes] lunghezza hash
  - Memory Size: **m** [8p – 2^{32} – 1 kilobytes]
  - Iterations: **t** [1-2^{32}-1 bytes] usato per configurare il tempo di esecuzione indipendentemente dall'occupazione di memoria
  - Version: **0x13**
  - Key: **K** [0 - 2^{32} – 1 bytes] lunghezza chiave (non necessaria)
  - Associated Data: **X** [0-2^{32}-1 bytes]
  - Hash type: 0=Argon2d, 1=Argon2i, 2=Argon2id

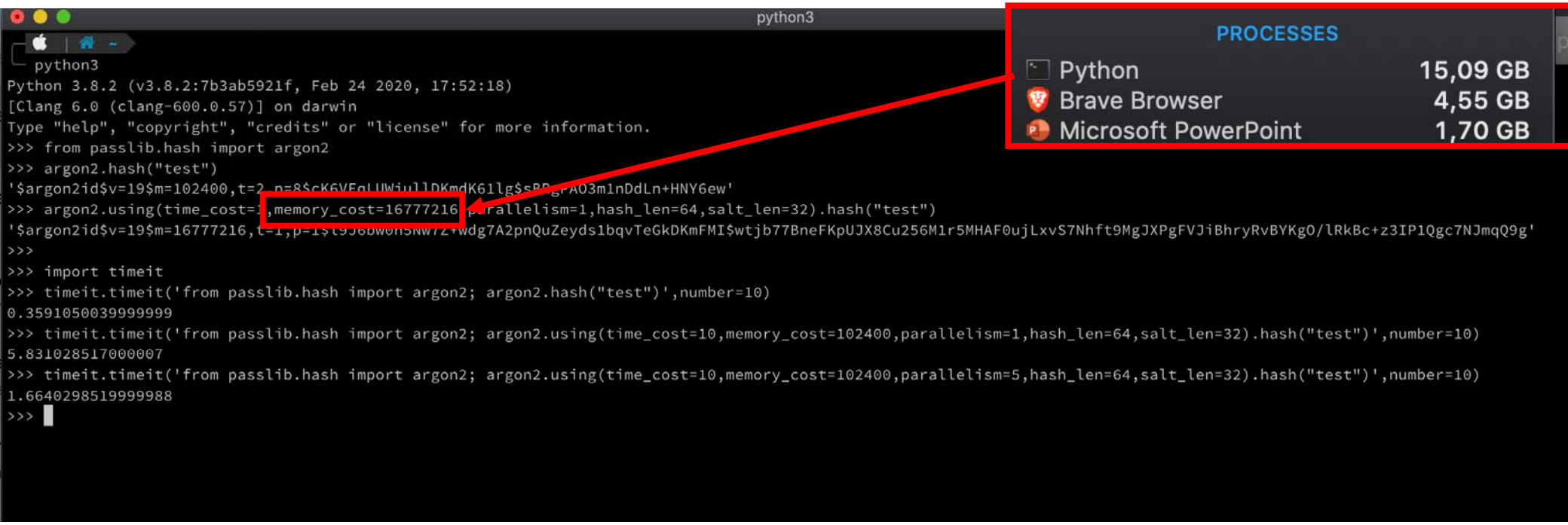
Argon2: processing

```
Generate initial 64-byte block H0.
 All the input parameters are concatenated and input as a source of additional entropy.
 Errata: RFC says H0 is 64-bits; PDF says H0 is 64-bytes.
 Errata: RFC says the Hash is H^, the PDF says it's H (but doesn't document what H is). It's actually Blake2b.
 Variable length items are prepended with their length as 32-bit little-endian integers.
buffer ← parallelism ǁ tagLength ǁ memorySizeKB ǁ iterations ǁ version ǁ hashType
      ǁ Length(password)       ǁ Password
      ǁ Length(salt)           ǁ salt
      ǁ Length(key)            ǁ key
      ǁ Length(associatedData) ǁ associatedData
H0 ← Blake2b(buffer, 64) //default hash size of Blake2b is 64-bytes

Calculate number of 1 KB blocks by rounding down memorySizeKB to the nearest multiple of 4*parallelism kibibytes
blockCount ← Floor(memorySizeKB, 4*parallelism)

Allocate two-dimensional array of 1 KiB blocks (parallelism rows x columnCount columns)
columnCount ← blockCount / parallelism;   //In the RFC, columnCount is referred to as q

Compute the first and second block (i.e. column zero and one ) of each lane (i.e. row)
for i ← 0 to parallelism-1 do for each row
   Bi[0] ← Hash(H0 ǁ 0 ǁ i, 1024) //Generate a 1024-byte digest
   Bi[1] ← Hash(H0 ǁ 1 ǁ i, 1024) //Generate a 1024-byte digest

Compute remaining columns of each lane
for i ← 0 to parallelism-1 do //for each row
   for j ← 2 to columnCount-1 do //for each subsequent column
      //i' and j' indexes depend if it's Argon2i, Argon2d, or Argon2id (See section 3.4)
      i', j' ← GetBlockIndexes(i, j)  //the GetBlockIndexes function is not defined
      Bi[j] = G(Bi[j-1], Bi'[j']) //the G hash function is not defined

Further passes when iterations > 1
for nIteration ← 2 to iterations do
   for i ← 0 to parallelism-1 do for each row
     for j ← 0 to columnCount-1 do //for each subsequent column
        //i' and j' indexes depend if it's Argon2i, Argon2d, or Argon2id (See section 3.4)
        i', j' ← GetBlockIndexes(i, j)
        if j == 0 then
          Bi[0] = Bi[0] xor G(Bi[columnCount-1], Bi'[j'])
        else
          Bi[j] = Bi[j] xor G(Bi[j-1], Bi'[j'])

Compute final block C as the XOR of the last column of each row
C ← B0[columnCount-1]
for i ← 1 to parallelism-1 do
   C ← C xor Bi[columnCount-1]

Compute output tag
return Hash(C, tagLength)
```

Source wikipedia

- Hash è una funzione interna basata su Blake2b:
  - τ <= 64 bytes:
    - Hash = Blake2b
  - τ > 64 bytes:
    - Hash = custom function ricavata da Blake2b

Argon2 e Passlib fun

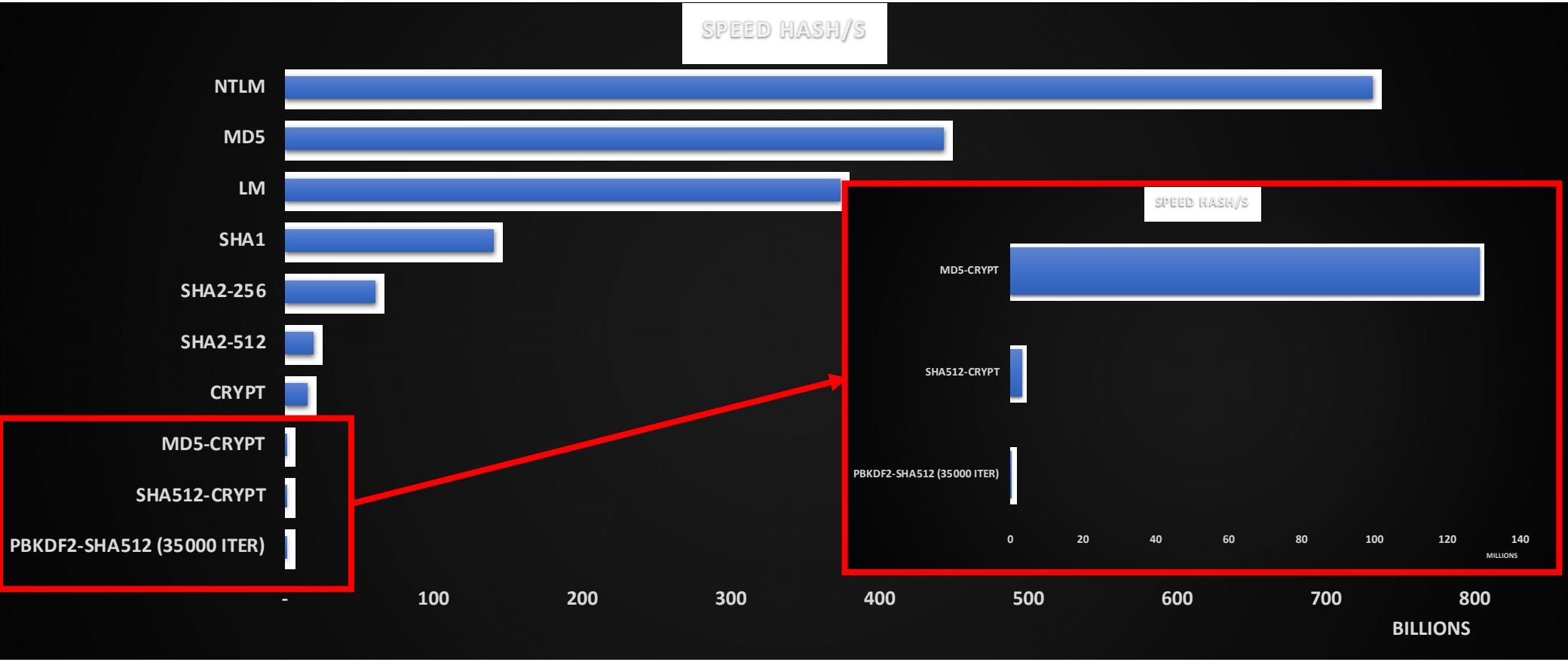


GPU cracking: esempio (1)

- GPU permettono di effettuare password guessing in tempi ridotti:
  - Sfrutta la capacità delle GPU di eseguire elaborazioni in parallelo;
  - Si parla di GPGPU (General Purpose GPU);
  - Utilizzate librerie come OpenCL;
- A Password^12 presentato:
  - Cluster 4 nodi
  - 25 schede grafiche AMD Radeon
  - Infinibend 4x SDR (8Gbit/s)
  - 7KW elettricità
  - Software: Oclhashcat-plus
    - (oggi hashcat)

GPU cracking: esempio (2)
Sistema in AWS con 8 NVIDIA Tesla V100

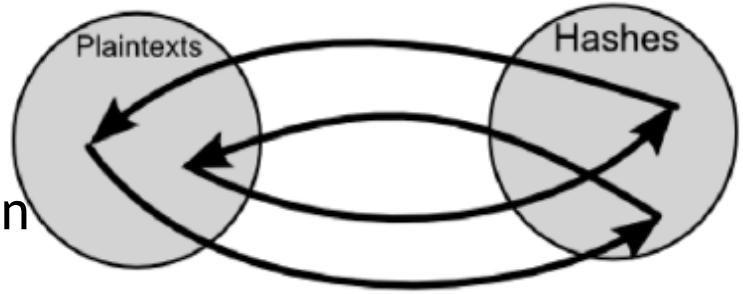
- 8x Tesla V100-SXM2-16GB
  - **Software**: Hashcat 5.1.0, Nvidia driver 418.67
  - **OS**: Ubuntu 18.04.2 LTS



Rainbow Tables

A **rainbow table** is a precomputed **table** for reversing cryptographic hash functions, usually for cracking password hashes. **Tables** are usually used in recovering a password (or credit card numbers, etc.) up to a certain length consisting of a limited set of characters.

- Le funzioni che gestiscono le tabelle sono le seguenti:
  - **Funzione Hash:** prende come argomento una password restituisce un hash.
  - **Funzione di Riduzione:** prende come argomento l'hash prodotto dalla precedente funzione e genera una password (vengono utilizzate n diverse funzioni di riduzione)

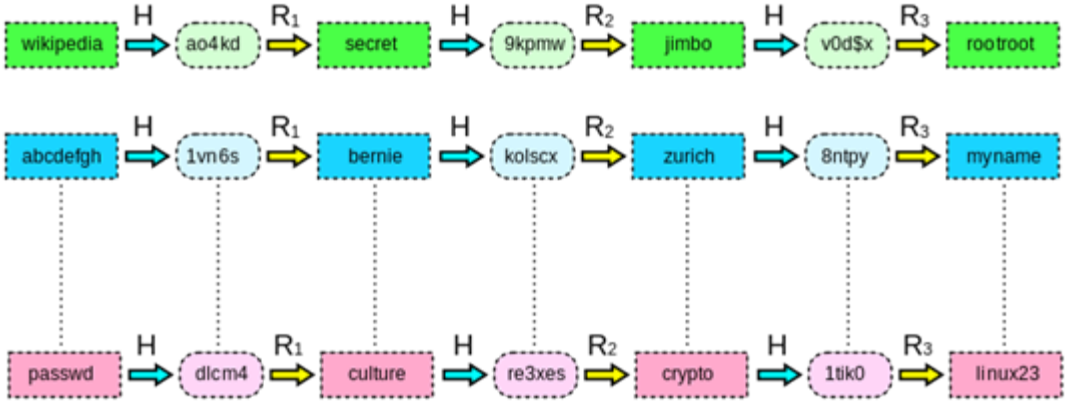


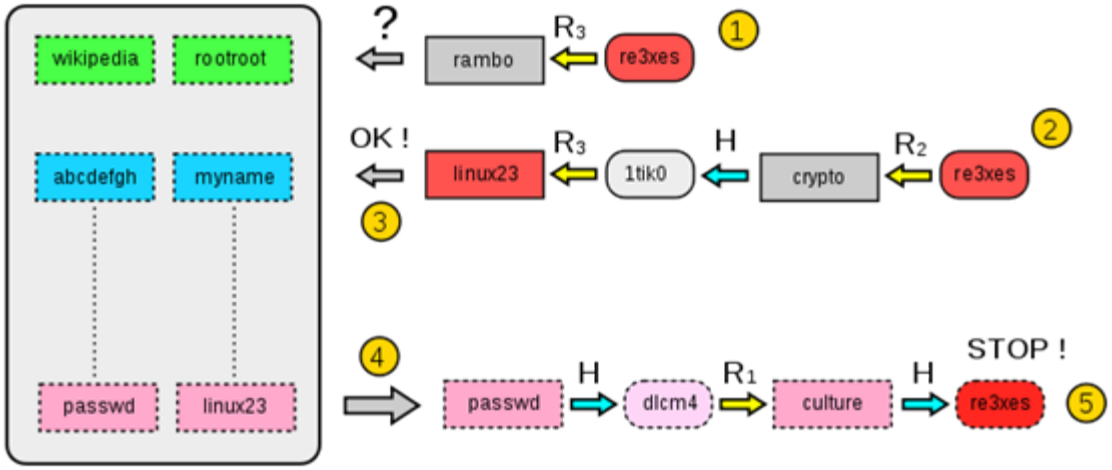
Rainbow Tables (2)

- Le rainbow tables sono costituite da un elenco di password che rappresentano l'inizio e la fine della catena:
  - Più la catena è lunga minore è lo spazio occupato su disco ma maggiore sarà la quantità di lavoro richiesta dalla CPU (e il tempo richiesto per il guessing).
  - Più la catena è corta, maggiore sarà lo spazio occupato su disco ma minore sarà la quantità di lavoro richiesta dalla CPU (e il tempo richiesto per il guessing).
- Le rainbow table rappresentano un compromesso tra la dimensione della tabella e il tempo richiesto per il guessing.

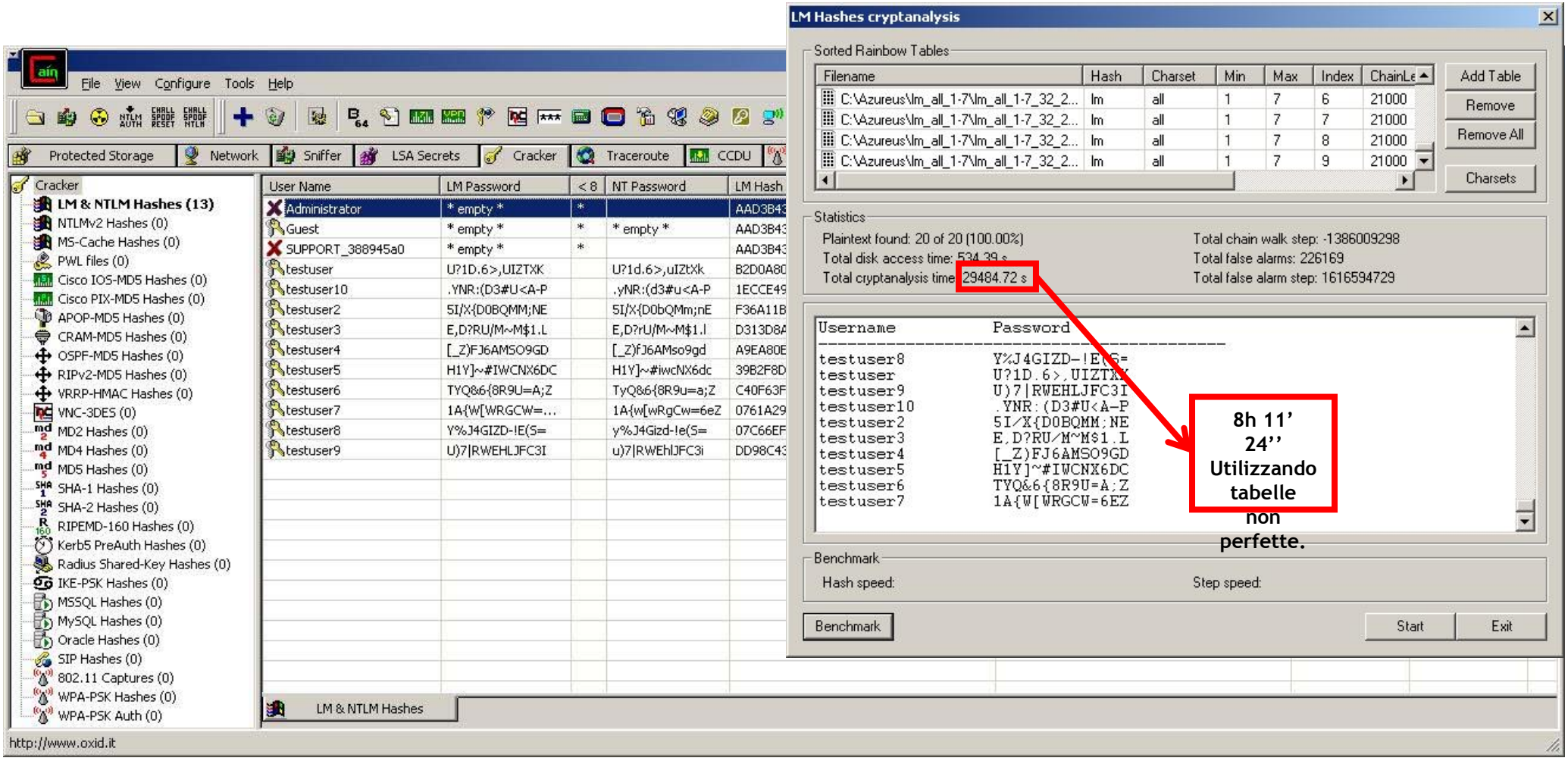
Rainbow Tables (3)

- Vediamo un esempio con 3 funzioni di riduzione:
  - Hash **re3xes**:
    - viene applicata **R3** ma la password prodotta non appare in nessuna fine della catena.
    - Viene allora applicata **R2 -> H -> R3** questa sequenza genera linux23 che appare alla fine di una catena.
    - Recupero la password iniziale e percorro la catena applicando **H->R1** ottenendo come password **culture**.
- Dove:
  - https://freerainbowtables.com/
  - https://ophcrack.sourceforge.io/tables.php





Rainbow Tables: esempio hash LM



Token: “what you have”

- I sistemi one-time password spesso si appoggiano su token hardware o su software specifici che generano la lista di password da inserire (spesso chiamati token software).
- Esistono due tipi di token hardware
  - **Token sincroni**: utilizzano come parte dell’algoritmo di generazione della password one-time un **timestamp**. Questo richiede che il token contenga un orologio e che questo sia sincronizzato con quello del server.
    - Si utilizza un meccanismo **a finestre temporali**. A ogni finestra è associato una password one-time.
  - **Token a contatore**: utilizzano come parte dell’algoritmo di generazione della password one-time **un contatore**. Questo richiede che contatori sul server e sul token siano sincronizzati.

Two-Factor Token

- Se il token hardware viene perso, chiunque ne entri in possesso può impersonare il proprietario legittimo.
- Per questo motivo spesso si utilizzano token two-factor (what you have & what you know). Si possono seguire tre approcci diversi:
  - **PIN accodato alla password one-time**
    - In questo caso il server dovrà verificare sia il PIN che il tokencode
  - **PIN come "ingrediente" per la password one-time**
    - In questo caso il tokencode è funzione anche del PIN.
    - È il server a doversi accorgere di PIN errati
  - **PIN come password del token**
    - In questo caso il PIN è immagazzinato nel token
    - È il token che si accorge di PIN errati e può disabilitarsi dopo n tentativi

RSA SecureID Key Fob

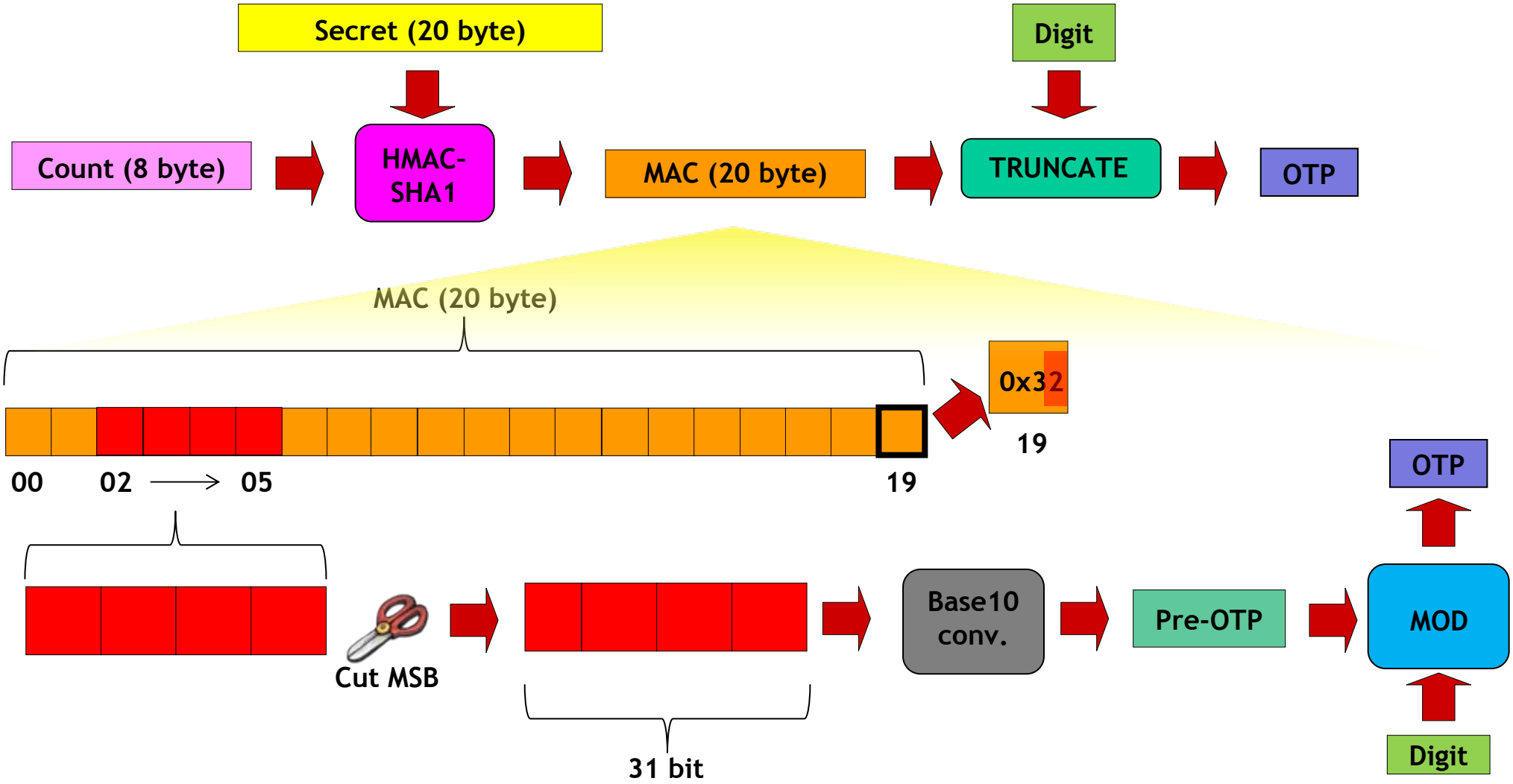
RSA SecureID PINpad


ActivCard Plus

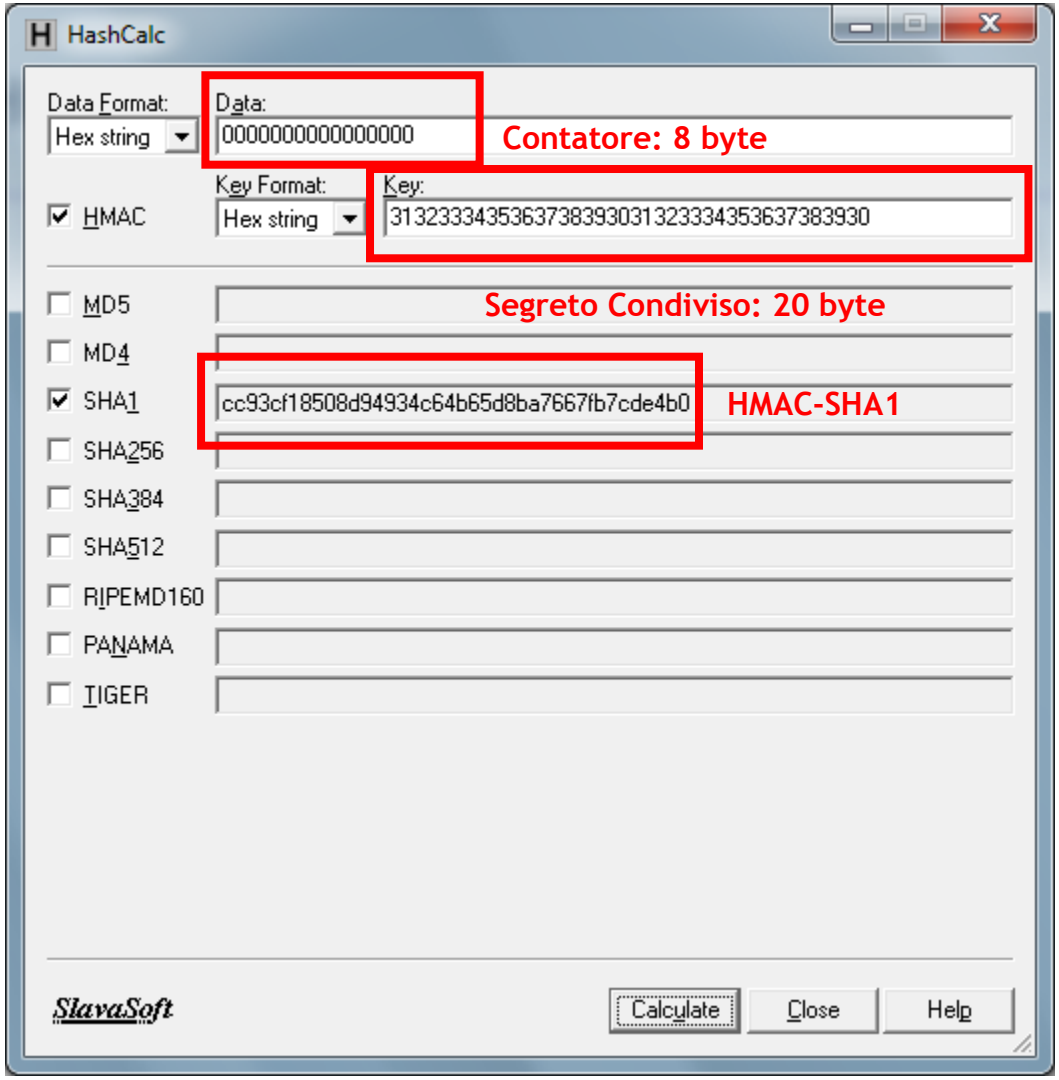
HOTP: HMAC Based One Time Password

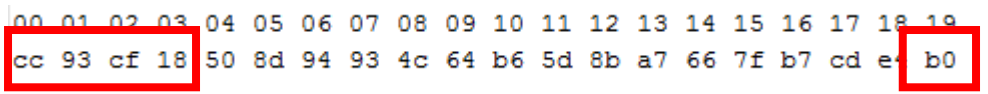
- **HOTP** (RFC 4226) specifica un algoritmo per la generazione di one time password utilizzando la funzione HMAC.
  - Permette **interoperabilità** tra diversi fornitori di software e hardware.
  - L'algoritmo proposto:
    - È utilizzabile in token a **contatore** e non in token sincroni;
    - È semplice da implementare in hardware a **basse capacità computazionali** e **basso consumo energetico**;
    - Non richiede dispositivi hardware dotati di tastiera o display ampio.
    - Genera OTP semplici da leggere ed inserire in una form di login.
    - Utilizza un segreto condiviso di almeno 128 bit (lo standard raccomanda un segreto di **almeno 160 bit**).
  - Utilizza **HMAC-SHA1** come funzione base per il calcolo dell'OTP
  - Permette di generare OTP di **lunghezza variable.**

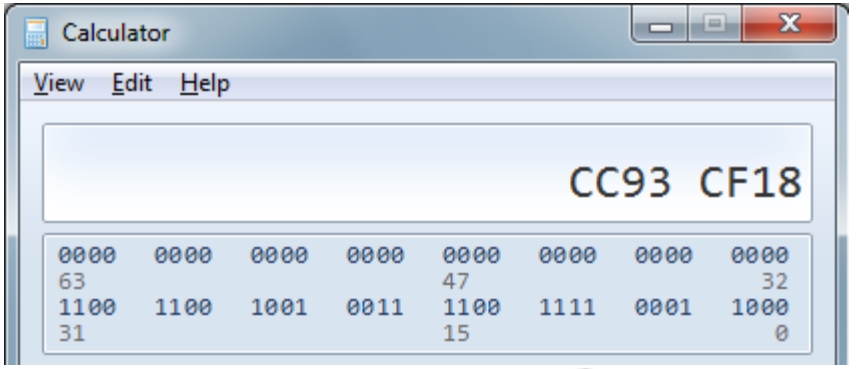
HOTP: Generazione OTP

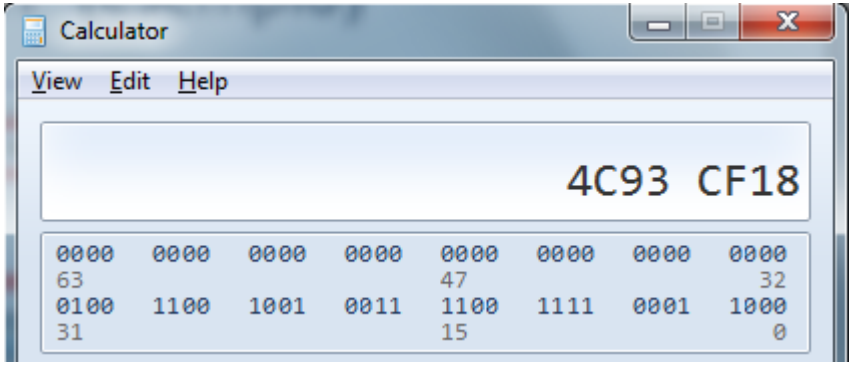


HOTP: calcolo OTP (esempio)



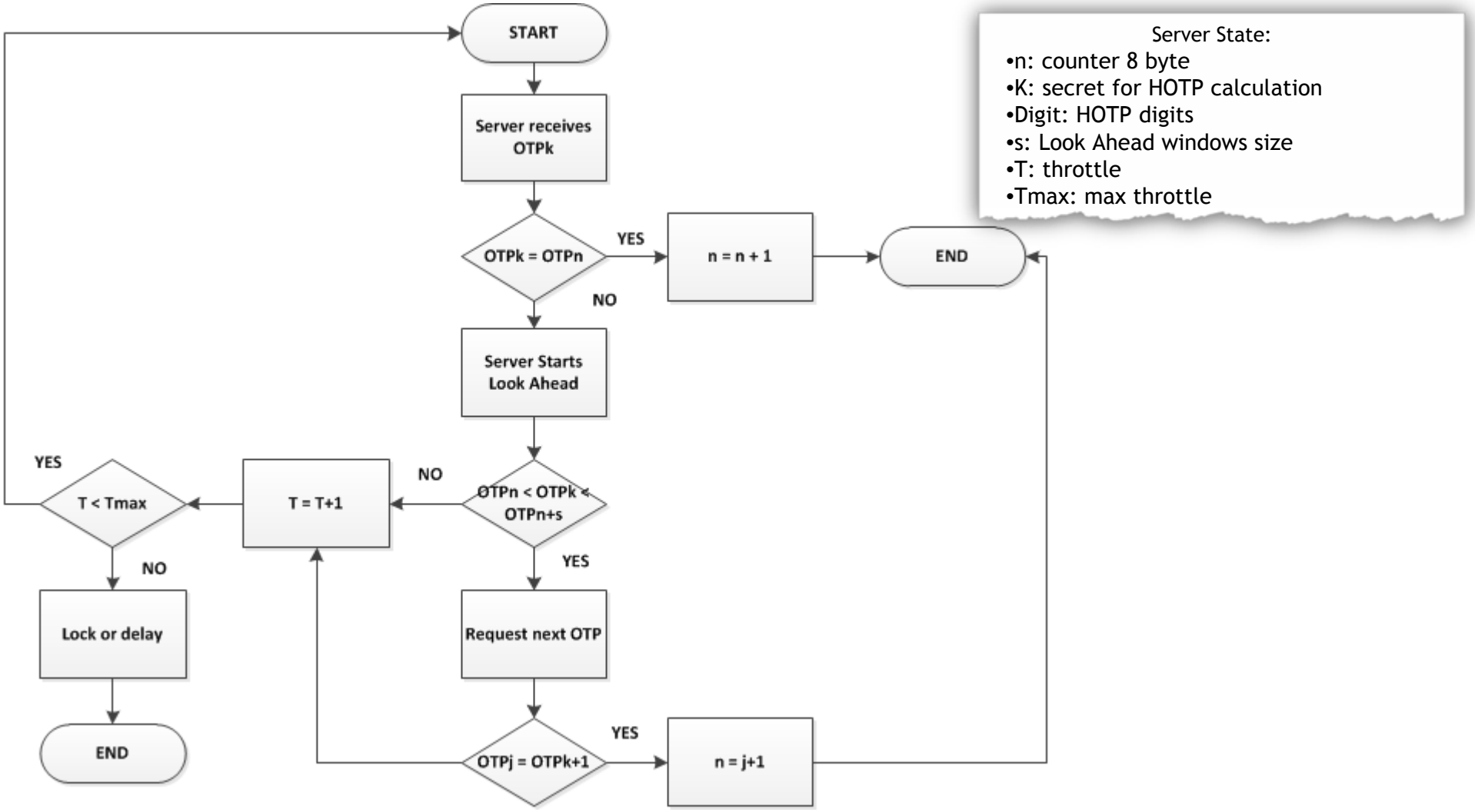






1.284.755.224 mod 10^6 = 755224

HOTP: Validazione OTP

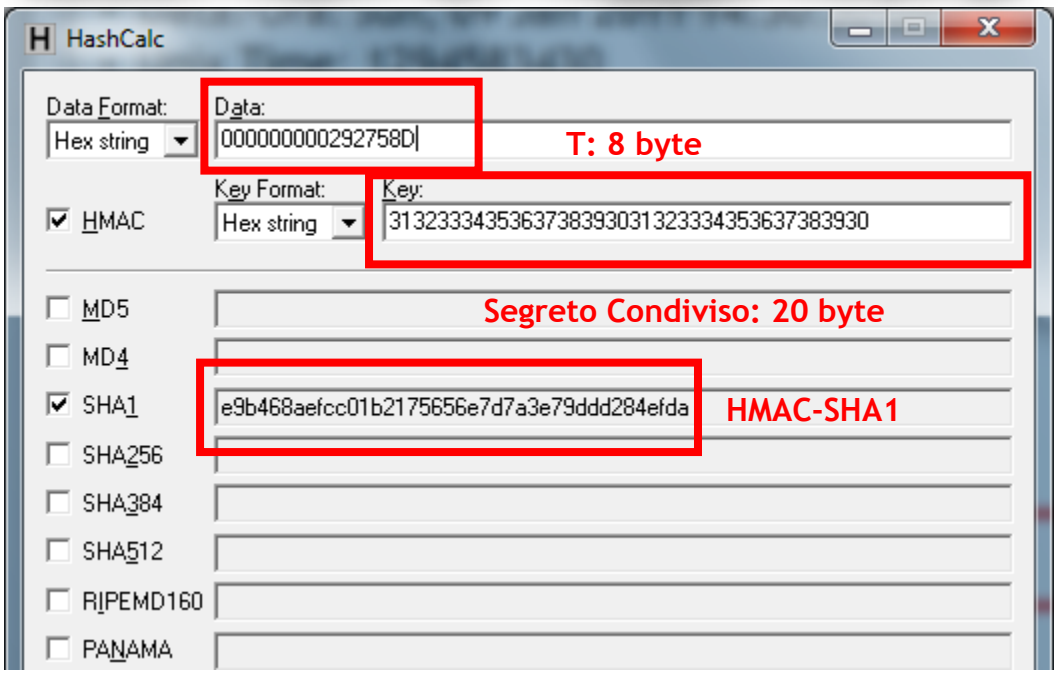


TOTP: Time-based One-time Password

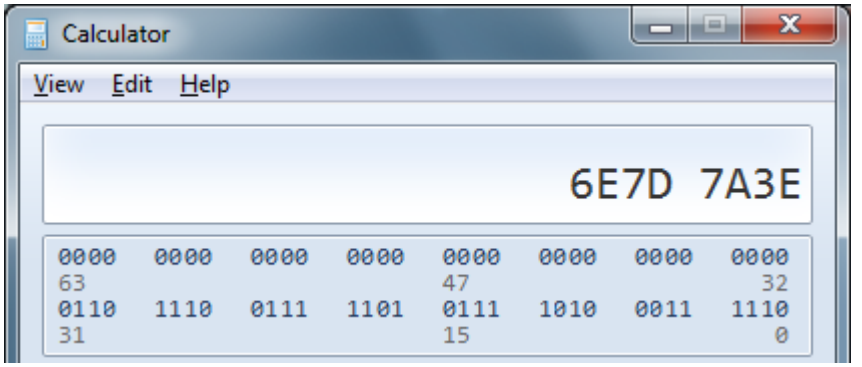
- **TOTP** è una proposta per un nuovo standard IETF che estende HOTP (RFC 4226) per la generazione di one time password utilizzando la funzione HMAC.
  - Permette **interoperabilità** tra diversi fornitori di software e hardware.
  - L'algoritmo proposto ha le medesime caratteristiche di HOTP:
    - Utilizza lo Unix Time: rappresenta il numero di secondi trascorsi dal 01 gennaio 1970 UTC (Coordinated Universal Time)
    - È utilizzabile in token **sincroni** e non in token a contatore;
  - La generazione della OTP utilizza lo stesso algoritmo visto per HOTP, ma utilizza un valore di 8 byte **dipendente dal tempo** al posto del contatore:
    - T (8 byte) = $\lfloor$(UnixTimeCorrente – T0)/X$\rfloor$
      - X: time step in secondi (default 30)
      - T0: unix time dal quale iniziare a contare (default 0).

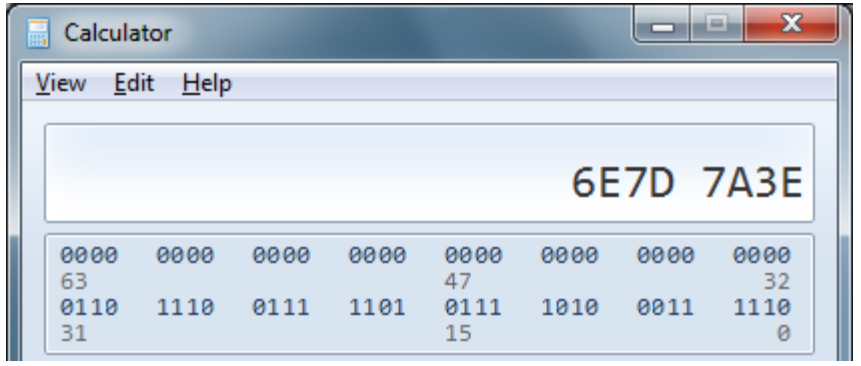
TOTP: calcolo OTP (esempio)

- Data/Ora: Sun, 09 Jan 2011 14:30:30 GMT
- Unix Time: 1294583430
- T0 = 0;
- X = 30;
- T = (1294583430 – 0)/30 = 43152781
- T = 0x0000 0000 0292 758D




```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19
e9 b4 68 ae fc c0 1b 21 75 65 6e 7d 7a 3e 79 dd d2 84 ef da
```





1.853.717.054 mod 10^6 = 717054

OCRA: OATH Challenge-Response Algorithm

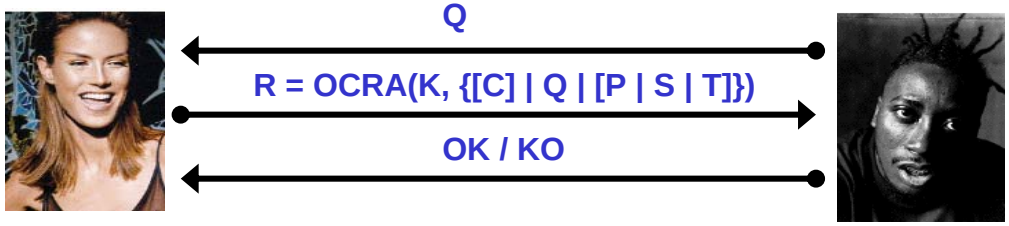
- **OCRA** è una proposta per un nuovo standard IETF per autenticazione challenge-response one time che utilizza HOTP (RFC 6287).
  - Permette **interoperabilità** tra diversi fornitori di software e hardware.
  - Sono supportate diverse modalità operative:
    - One-Way Challenge-Response
    - Mutual Challenge-Response
    - Plain Signature
    - Signature with Server Authentication
  - La generazione della OTP utilizza lo stesso algoritmo visto per HOTP ma utilizza un valore di lunghezza variabile:
    DataInput = {OCRASuite | 00 | C | Q | P | S | T}

OCRA: OATH Challenge-Response Algorithm

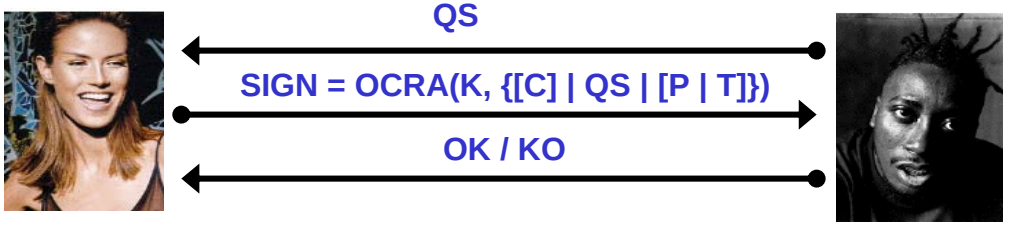
- Di seguito gli elementi di DataInput ([ ] = facoltativo):
  - **OCRASuite**: indica come calcolare il token OCRA:
    - OCRASuite: <Algorithm>:<CryptoFunction>:<DataInput>
      - <Algorithm>: OCRA-1 o OCRA-2
      - <CryptoFunction>: HOTP-SHA1-t o HOTP-SHA256-t o HOTP-SHA512-t
      - <DataInput>: A = alfanum, N = num, H = hex
  - **00**: byte nullo separatore [1 byte]
  - **[C]**: contatore [8 byte]
  - **Q**: challenge [128 byte] pad con 0 se più corto
  - **[P]**: hash (sha-1 [20 byte], sha-256 [32 byte] o sha-512 [64 byte]) di un PIN o password condivisi.
  - **[S]**: stringa codificata in UTF-8 [max 512 byte]
  - **[T]**: time step [8 byte]

OCRA: Modalità Operative

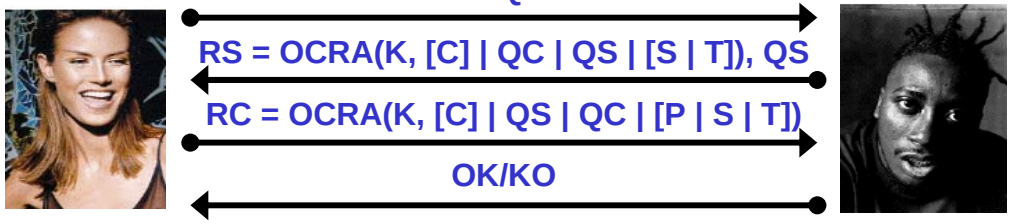
One Way Challenge Response



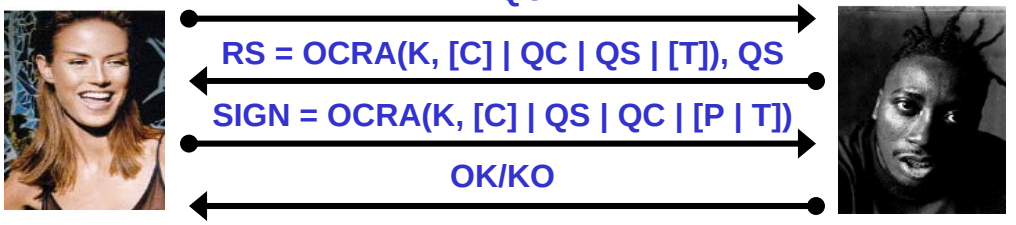
Plain Signature



Mutual Challenge Response



Signature with Server Authentocation

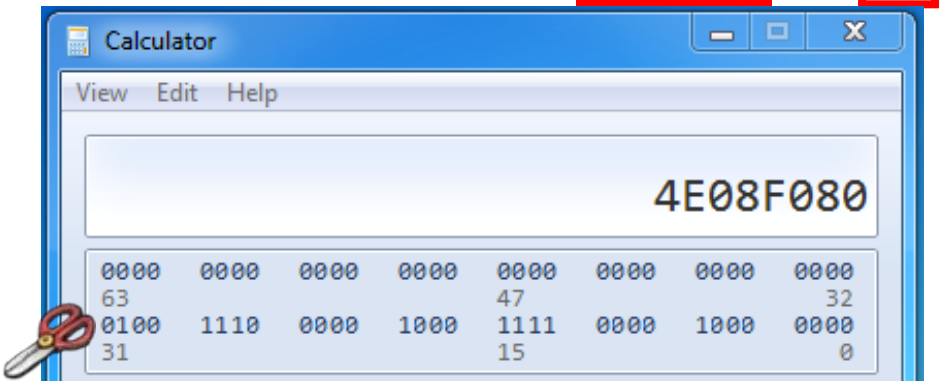


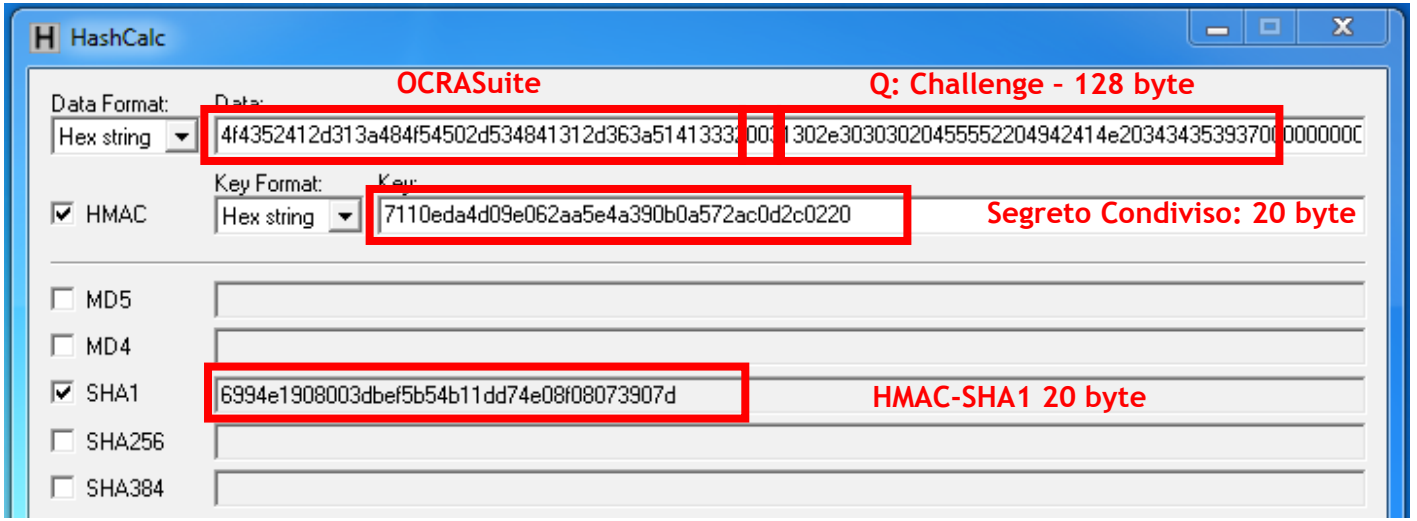
OCRA: esempio

- OCRASuite: **OCRA-1:HOTP-SHA1-6:QA32**
- Q:**10.000 EUR IBAN 44597**
- C = null;
- P = null;
- S = null;
- T = null;

DataInput = {OCRASuite|00|Q }

```
00 01 02 03 04 05 06 07 08 09 10 11 12 13 15 15 16 17 18 19
69 94 e1 90 80 03 db ef 5b 54 b1 1d d7 4e 08 f0 80 73 90 7d
```





1.309.208.704 mod 10^6 =

208704

Funzioni di Hash Crittografiche

- Una **funzione di hash** è una funzione H che, dato un input M di dimensione qualsiasi, produce un output h (digest) di **dimensione fissa**.
- La funzione di hash genera **un'impronta** dell'input processato (o messaggio).
  - Poiché l'input ha lunghezza **variabile** mentre l'output ha lunghezza **fissa** (qualche centinaia di bit in genere) é possibile che messaggi diversi generino lo stesso message digest (**collisione**).
  - Ovviamente non è detto che sia computazionalmente praticabile riuscire a trovare tali collisioni.

Proprietà funzioni crittografiche di hash

- Una funzione di hash crittografica deve possedere le seguenti 5 proprietà:
  1. É una funzione **deterministica**: lo stesso input genera sempre lo stesso output.
  2. È molto **veloce** da calcolare per qualunque dimensione dell'input.
  3. È impossibile ottenere il messaggio dal digest se non provando tutti i possibili messaggi (**unidirezionalità**)
  4. Un **piccolo** cambiamento nel messaggio deve modificare i digest in modo tale che i due messaggi in input appaiano come **non correlati**.
  5. È impossibile (computazionalmente costoso) trovare **due messaggi** diversi che generino lo **stesso** digest.

Sicurezza funzioni crittografiche

- La sicurezza di una funzione di hash viene valutata in base alle seguenti caratteristiche:
  - *pre-image resistance*: dato un valore di hash h, deve essere difficile risalire ad un messaggio m con hash(m) = h.
  - *second pre-image resistance*: dato un input *m1*, deve essere difficile trovare un secondo input *m2* tale che *hash(m1) = hash(m2)*
  - *collision resistance: d*ati due messaggi *m1* ed *m2,* deve essere difficile che i due messaggi abbiano lo stesso hash, quindi con hash(m1) = hash(m2)
    - La complessità del secondo e del terzo attacco è molto diversa! Se l'hash è lungo m bit, la complessità del primo è 2^m, quella del secondo è 2^(m/2).

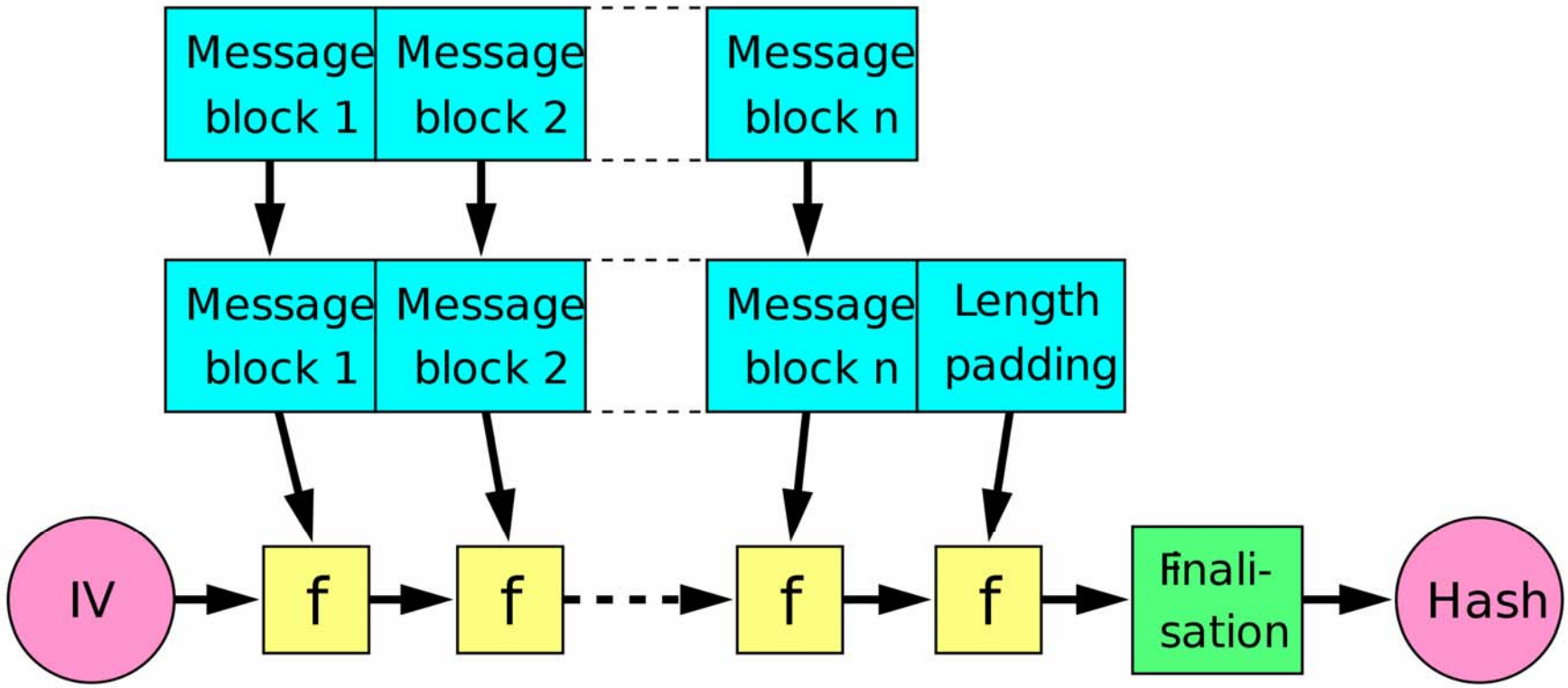
La costruzione di Merkle-Damgård

- È un metodo per costruire delle funzioni crittografiche di hash **resistenti alle collisioni** utilizzando delle funzioni di compressione one-way.
  - è stata utilizzata nell'implementazione di molti algoritmi di hash, come ad esempio **MD5**, **SHA1** e **SHA2**.
- Le funzioni di hash MD applicano **padding** al messaggio in modo che esso sia un multiplo di una lunghezza fissa (e.g. 512 bit)
  - Le funzioni di compressione non supportano input a lunghezza variabile.

La costruzione di Merkle-Damgård (2)

- La funzione di compressione f:
  - Prende in input **due quantità** ed restituisce in output una quantità di **lunghezza uguale** a quella di uno dei due input.
  - La costruzione parte con un **vettore di inizializzazione** (IV) che dipende dalla specifica implementazione.
- Per ogni blocco del messaggio la funzione di compressione prende input il **risultato** processato fino a quel momento e lo combina con il **blocco** producendo un risultato intermedio.
  - Sull'ultimo blocco viene effettuato **padding**.

La costruzione di Merkle-Damgård (3)



*Courtesy of Wikipedia

La costruzione di Merkle-Damgård (4)

- Prima di iniziare l'elaborazione, si aggiunge al messaggio un **padding** in modo che la lunghezza totale risulti un multiplo della lunghezza del blocco (es: 512 bit):
  - si aggiunge un bit a 1 e poi tanti bit a 0 quanto basta perché la lunghezza risulti di 64 bit minore rispetto a un multiplo di 512 bit (se la lunghezza originale è già corretta si effettua ugualmente padding);
  - si aggiungono 64 bit contenenti la lunghezza originale del messaggio.

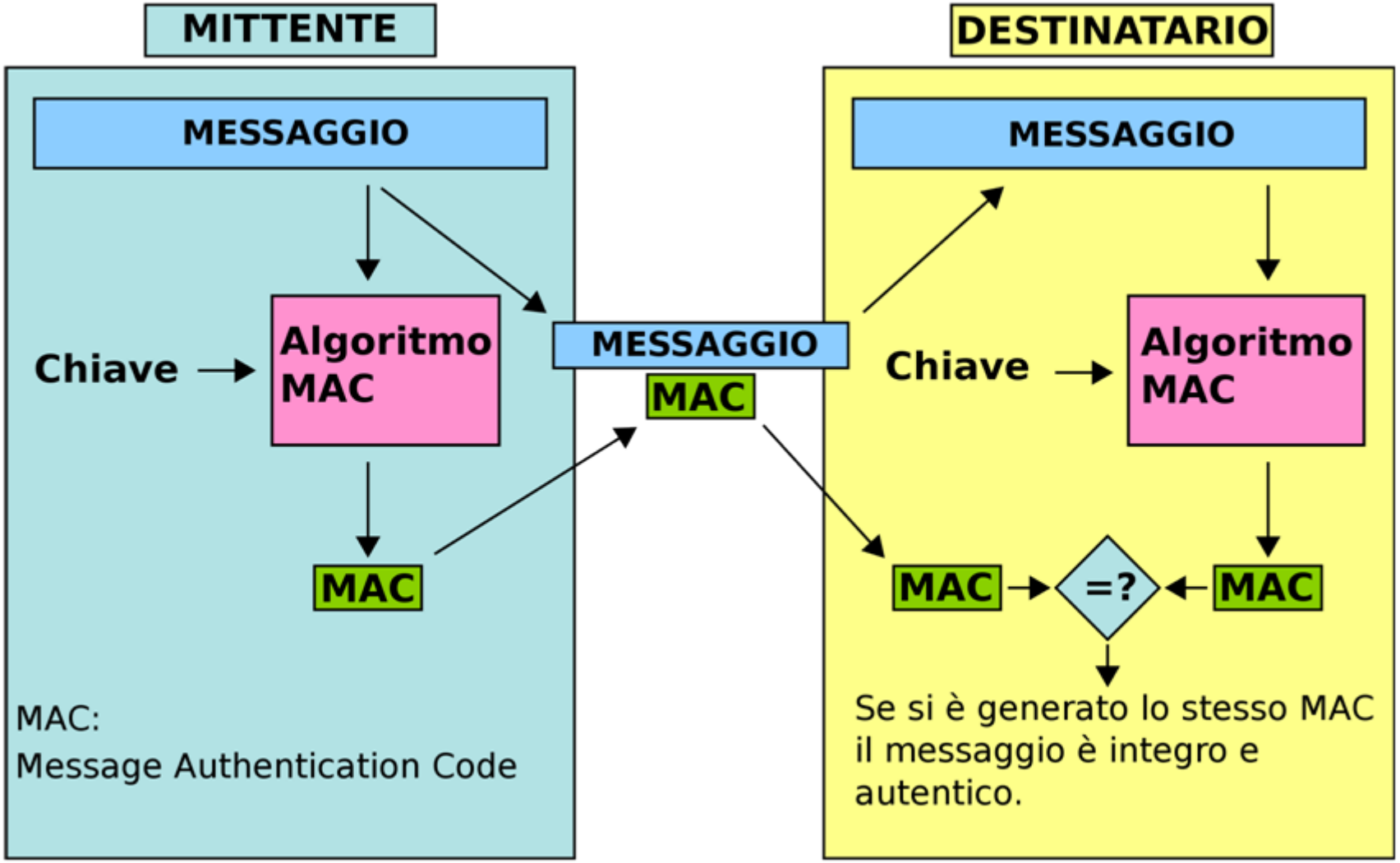
La costruzione di Merkle-Damgård (5)

- Esempio:
  - Blocco:
    - 01100001 01100010 01100011 01100100 01100101
  - Si appende un bit a 1
    - 01100001 01100010 01100011 01100100 01100101 1
  - Si appendono bit 0 fin quando il blocco non è lungo 512 – 64 = 448 bit. Nel nostro caso abbiamo 41 bit nel blocco, quindi dobbiamo aggiungere 512 – 64 – 41 = 407 bit a 0.
    - 61626364 65800000 00000000 00000000
    - 00000000 00000000 00000000 00000000
    - 00000000 00000000 00000000 00000000
    - 00000000 00000000
  - Si appendono 64 bit rappresentanti la lunghezza del messaggio (40 == 0x28)
    - 61626364 65800000 00000000 00000000
    - 00000000 00000000 00000000 00000000
    - 00000000 00000000 00000000 00000000
    - 00000000 00000000 00000000 00000028

MAC: Message Authentication Code (1)

- Un **message authentication code** (MAC) è un piccolo blocco di dati utilizzato per garantire l'autenticazione e integrità di un messaggio.
- Un algoritmo MAC accetta in ingresso: un **segreto** e un **messaggio** da autenticare di lunghezza arbitraria.
- In ricezione il destinatario opererà in maniera identica sul messaggio pervenuto in chiaro ricalcolando il MAC con lo stesso algoritmo e lo stesso segreto: se i due MAC coincidono si ha **autenticazione** e **integrità** del messaggio inviato.

MAC: Message Authentication Code (2)



MAC con le funzioni di HASH?

- Le funzioni di HASH sono molto veloci in software, e producono ”un’impronta” del file.
- Se posso introdurre come ingrediente della funzione di hash anche un segreto, posso facilmene ottenere un MAC rudimentale.
- Allora perché non usare un keyed hash come MAC?
  - **MAC = H(secret||data)**
- Se utilizziamo questo approccio un attaccante può facilmente estendere il messaggio e ricalcolare l’hash **senza** conoscere il segreto Allora

Hash length extension attack

- In un attacco di hash extension l'attaccante utilizza:
  - **H(message1)**: digest di un messaggio non noto.
  - **len(message1)**: la lunghezza del messaggio stesso
- E calcola H(message1||message2) con message2 controllato dall'attaccante stesso.
- Nel caso si utilizzi un hash di **Merkle-Damgård** come MAC:
  - MAC = h(secret||message)
  - Se len(secret||message) è nota
- Allora è possibile estendere in messaggio e ricalcolare il MAC **senza conoscere il segreto** (secret)

Hash length extension attack (2)

- Esempio:
  - Hash: **sha1**
  - Segreto: **$3cr3t!1#** (non noto all'attaccante)
  - Dati: **admin=0**
  - Message: **$3cr3t!#1admin=0**
  - Sha($3cr3t!1#||admin=0):
    - 63f8aceadc264ee6e8495ff18bf651a092f522ee
- L'attaccante può appendere al messaggio |admin=1
  - Nuovi dati: **admin=0|admin=1**
  - L'attaccante non deve fare altro che fornire l'**hash calcolato precedentemente** nello **stato della funzione di hash** e continuare il lavoro lì dove era stato lasciato.

Hash length extension attack (3)

- Il nuovo messaggio dovrà includere il padding che l'algoritmo aggiungerebbe, in modo che i nuovi dati (|admin=1) siano allineati a un nuovo blocco.
- Quanto padding bisogna aggiungere?
  - Il messaggio é di 16 byte
  - Il blocco in totale deve essere lungo 64 byte (512 bit)
  - 64 – 16 = 48 byte di padding
    - Il primo byte é settato a \x80 (1000 0000)
    - Gli ultimi 8 byte sono settato alla lunghezza del messaggio originale (16 byte): \x00\x00\x00\x00 \x00\x00\x00\x80
    - Quindi il padding sarà:
      - \x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80

Hash length extension attack (4)

- Il messaggio modificato sarà:
  - admin=0\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80|admin=1
  - E il nuovo mac calcolato senza conoscere il segreto:
    - 4145ed66a63710d1871279127ebb4f5cd05fd1ad
  - Si può facilmente verificare che:
    - Sha1($3cr3t!1#admin=0\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80|admin=1) ==

      4145ed66a63710d1871279127ebb4f5cd05fd1ad

Hash length extension attack (5)

```python
#!/usr/bin/python2
import hlextend
import hashlib
import binascii

secret = '$3cr3t!1#'
data = 'admin=0'
data_append = '|admin=1'

mac = hashlib.sha1(secret + data).hexdigest()
secret_size = len(secret)

print '[*] Initial MAC: {0}'.format(mac)
print '[*] Initial Data: {0}'.format(data)
print '[*] Data to append: {0}'.format(data_append)

sha1 = hlextend.sha1()
updated_message = sha1.extend(data_append, data, secret_size, mac, raw = True)
updated_mac = sha1.hexdigest()

print '[*] The new message is: {0}'.format(updated_message.encode("string_escape"))
print '[*] The new message hex is: {0}'.format(binascii.hexlify(updated_message))
print '[*] The new MAC is: {0}'.format(updated_mac)
```

Hash length extension attack (6)

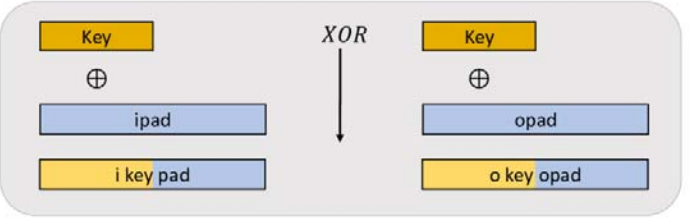
```
1. bash
M062761RDJGH6:anonimages antonio_forzieri$ python hash_extend_example.py
[*] Initial MAC: 63f8aceadc264ee6e8495ff18bf651a092f522ee
[*] Initial Data: admin=0
[*] Data to append: |admin=1
[*] The new message is: admin=0\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x80|admin=1
[*] The new message hex is: 61646d696e3d3080000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000807c61646d696e3d31
[*] The new MAC is: 4145ed66a63710d1871279127ebb4f5cd05fd1ad
M062761RDJGH6:anonimages antonio_forzieri$
```

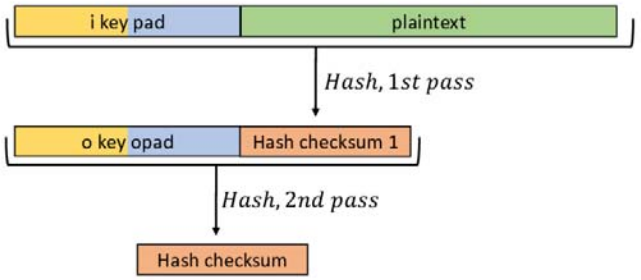
Keyed-Hashing for Message Authentication

- **HMAC** è una proposta informational di IETF (**RFC2104**)
- Può essere utilizzato con ogni funzione di hash (e.g. **MD5**, **SHA1**)
- Utilizza come ulteriore ingrediente un segreto condiviso.
- HMAC si pone i seguenti obettivi:
  - Utilizzare senza alcuna **modifica** funzioni di hash esistenti. In particolare funzioni molto performanti in software
  - Preservare le **performance** della funzione di hash utilizzata senza degradarne le performance
  - Utilizzare le **chiavi** in maniera molto semplice
  - Essere semplice da **analizzare** da un punto di vista crittografico
  - Permettere il cambio di funzone di hash utilizzata in maneira semplice.

HMAC: funzionamento

1. Si appendono **0** a **K** in modo da creare una stringa di **B** bytes.
2. Si effettua lo **XOR** della stringa calcolata in 1 con ipad
3. Si concatena il testo ai **B** byte risultanti dallo **XOR** allo step **2**
4. Si effettua l'hash (H) della stringa risultante dallo step 3
5. Si effettua lo **XOR** della stringa ottenuta allo step 1 con opad
6. Si concatena la stringa ottenuta allo step 4 con i B byte dello step 5
7. Si applica la funzione di hash allo stream dello step 6 e si ottiene il risultato





- **H**: hash function
- **K**: secret
- **B**: block size for H (byte)
- **L**: digest length
- **ipad**: 0x36 x B times
- **opad**: 0x5C x B times

HMAC: proviamo ad implementarlo

Users > antonio_forzieri > Desktop > hmac_test.py > ...

```python
import hashlib
import hmac

### Impostiamo i parametri del nostro HMAC
hash_algo = "md5"
B = 64
message = b'what do ya want for nothing?'
key = b'Jefe'

### hash() for lazy people
def hash(M):
    H = hashlib.new(hash_algo)
    H.update(M)
    b_hash = H.digest()
    return b_hash

### XOR Translation tables
trans_5C = bytes((x ^ 0x5C) for x in range(256))
trans_36 = bytes((x ^ 0x36) for x in range(256))

### STEP1: Si appendono 0 a K in modo da creare una stringa di B bytes.
K_zpad=key.ljust(B,b'\0')
print("[*] Fine STEP1. K_zpad è lunga {0} byte".format(len(K_zpad)))

### STEP2: Si effettua lo XOR  della stringa calcolata in 1 con ipad
K_ipad=K_zpad.translate(trans_36)
print("[*] Fine STEP2: [Key XOR ipad] vale: 0x{0}".format(K_ipad.hex()))

### STEP3: Si concatena il testo ai B byte risultanti dallo XOR allo step 2

part1=K_ipad+message
print("[*] Fine STEP3: [(key XOR ipad) || message] vale: 0x{0}".format(part1.hex()))

### STEP4: Si effettua l'hash (H) della stringa risultante dallo step 3
hash1=hash(part1)
print("[*] Fine STEP4: [ H(key XOR ipad || message) ]: 0x{0}".format(hash1.hex()))

### STEP5: Si effettua lo XOR della stringa ottenuta allo step 1 con opad
K_opad=K_zpad.translate(trans_5C)
print("[*] Fine STEP5: [key XOR opad] vale: 0x{0}".format(hash1.hex()))

### STEP6: Si concatena la stringa ottenuta allo step 4 con i B byte dello step 5
part2 = K_opad + hash1
print("[*] Fine STEP6: [(key XOR opad || H((key XOR ipad)|message ))] vale: 0x{0}".format(part2.hex()))

### STEP7: Si applica la funzione di hash allo stream dello step 6 e si ottiene il risultato
hash_final = hash(part2)
print("[*] Fine STEP7: [H(key XOR opad || H((key XOR ipad)|message )))] vale: 0x{0}".format(hash_final.hex()))

# Verifichiamo se i cnti tornano
check=hmac.new(key,message,digestmod=hash_algo)
print("[*] HMAC calcolato dalla libreria python vale: 0x{0}".format(check.hexdigest()))

if (check.hexdigest() == hash_final.hex()):
    print("[*] HMACs match: WELL DONE!")
else:
    print("[*] YOU LOSE!")
```

```
~/Desktop
cd /Users/antonio_forzieri/Desktop ; env /Library/Frameworks/Python.framework/Versions/3.8/bin/python3 /Users/antonio
s/antonio_forzieri/Desktop/hmac_test.py
[*] Fine STEP1. K_zpad è lunga 64 byte
[*] Fine STEP2: [Key XOR ipad] vale: 0x7c5350533636363636363636363636363636363636363636363636363636363636363636363636363636
[*] Fine STEP3: [(key XOR ipad) || message] vale: 0x7c5350533636363636363636363636363636363636363636363636363636363636363636363636363
[*] Fine STEP4: [H(key XOR ipad || message) ]: 0xc3db14c065f55203b033c81a697b97c5
[*] Fine STEP5: [key XOR opad] vale: 0xc3db14c065f55203b033c81a697b97c5
[*] Fine STEP6: [(key XOR opad || H((key XOR ipad)|message ))] vale: 0x16393a395c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c5c
[*] Fine STEP7: [H(key XOR opad || H((key XOR ipad)|message )))] vale: 0x750c783e6ab0b503eaa86e310a5db738
[*] HMAC calcolato dalla libreria python vale: 0x750c783e6ab0b503eaa86e310a5db738
[*] HMACs match: WELL DONE!
~/Desktop
```

- L'implementazione non considera il caso in cui la chiave sia più lunga di B byte.
- Serve solo per validare la comprensione di HMAC.

Antonio Forzieri
Splunk Inc

aforzieri@splunk.com
+393477819020