

# Introduction to cryptology (GBIN8U16)



## Password Hashing

Pierre Karpman

`pierre.karpman@univ-grenoble-alpes.fr`

`https://www-ljk.imag.fr/membres/Pierre.Karpman/tea.html`

2020-04-29

# Motivation: How to store a password?

---

A simple login/password interaction:

- 1 User  $U$  wants to log on system  $S$ ; sends password  $p$
- 2 System  $S$  checks password associated with  $U$  in database  $D = \{(U_i, p_i)\}$ ; grants access if equal to  $p$

A simple total break:

- 1 Adversary  $A$  steals database  $D$  (Quite realistic; happens a lot)

⇒ Passwords must never be stored *in clear*!

# How to solve this? With Crypto!

---

A first attempt (aborted):

- ▶ Store  $p$  encrypted with, say, CBC-ENC
- ▶  $U, S$  Need to store/know the user-dependent secret key:  
nothing is solved

A first attempt:

- ▶ Store  $p$  encrypted with, say, RSA-OAEP
- ▶  $U$  needs to know  $S$ 's public key
- ▶  $S$  has a single secret to store (but always used to decrypt; still bad)

# Hash functions to the rescue

---

A second attempt: go keyless!

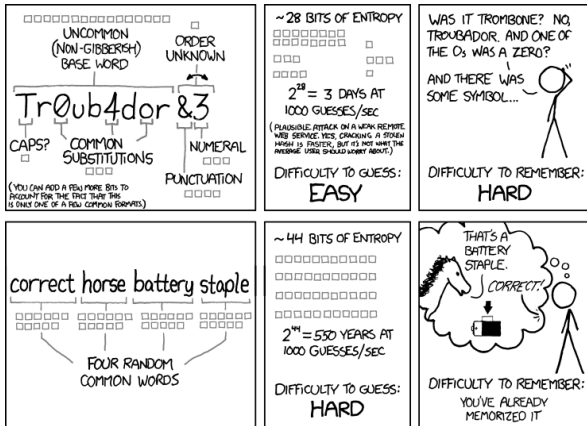
- ▶ Store hashed passwords  $\mathcal{H}(p) \rightsquigarrow D = \{(U_i, \mathcal{H}(p_i))\}$
- ▶  $S$  checks that the received password hashes to the right value
- ▶  $\Rightarrow$  similar to the simple hash-based PoD scheme! (If the communication channel and computations are secure, no need to reset the challenge)
- ▶ If  $\mathcal{H}$  is preimage-resistant,  $\mathcal{H}(p) \nrightarrow p$ ?
- ▶ Basically sound, but the security analysis is not so simple

# Passwords are not random

---

- ▶ Let  $\mathcal{H} : \{0, 1\}^* \rightarrow \{0, 1\}^n$ . For any explicit set  $\mathcal{S}$ ,  $\#\mathcal{S} \lesssim 2^{n/2}$ ,  $x \in \mathcal{S}$  can be found in time  $\leq \#\mathcal{S}$  given  $\mathcal{H}(x)$  (Question: why? how?)
- ▶ If  $\mathcal{H}(x)$  is used to identify  $x$ , any preimage works
- ▶ “Inverting”  $\mathcal{H}$  takes time  $\approx \min(2^n, \#\mathcal{S})$  (Assuming  $x \xleftarrow{\$} \mathcal{S}$ )
- ▶ Not a problem of hash functions specifically, just the absence of (other) secret

# Password entropy: a global issue



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

<https://xkcd.com/936/>

# So, What hash function to use?

---

Microsoft's LM hash? (1980's)

- 1 Truncate  $p$  to 14 ASCII characters
- 2 Convert it to uppercase
- 3 Split it in two halves  $p_0, p_1$
- 4  $\text{LMHash}(p) = \text{DES}(p_0, c) \parallel \text{DES}(p_1, c)$  for a fixed constant  $c$ 
  - ▶  $\text{DES} : \{0, 1\}^{56} \times \{0, 1\}^{64} \rightarrow \{0, 1\}^{64}$  is a block cipher

What's wrong with that?

- ▶ The two halves of the hash are processed separately
- ▶ Only  $69^7 \lesssim 2^{43}$  possible inputs per half
  - ▶ Only  $2^{20}$  seconds on one core of this laptop needed to exhaust them; time-memory tradeoffs are available
- ▶ *Impossible* to securely store a strong password

# So, What hash function to use?

---

Microsoft's LM hash? (1980's)

- 1 Truncate  $p$  to 14 ASCII characters
- 2 Convert it to uppercase
- 3 Split it in two halves  $p_0, p_1$
- 4  $\text{LMHash}(p) = \text{DES}(p_0, c) \parallel \text{DES}(p_1, c)$  for a fixed constant  $c$ 
  - ▶  $\text{DES} : \{0, 1\}^{56} \times \{0, 1\}^{64} \rightarrow \{0, 1\}^{64}$  is a block cipher

What's wrong with that?

- ▶ The two halves of the hash are processed separately
- ▶ Only  $69^7 \lesssim 2^{43}$  possible inputs per half
  - ▶ Only  $2^{20}$  seconds on one core of this laptop needed to exhaust them; time-memory tradeoffs are available
- ▶ *Impossible* to securely store a strong password



## A better choice: an actual hash function

---

- ▶ A “modern” answer: just take  $\mathcal{H}$  to be, say, SHA3-256
- ▶ Problem: multi-target attacks are (still) easy
  - ▶ An adversary may want to find one password among  $N$
  - ▶ For every candidate  $p'$ , check if  $\mathcal{H}(p') \in D$
  - ▶ The work is decreased by a factor  $\approx N$
  - ▶  $N$  might be large (say,  $> 1000$ )
- ▶ One counter-measure: use different functions for every user
  - ▶ Simple to implement: every user  $U_i$  selects a large random number  $r_i$  (the “salt”);  $D = \{(U_i, r_i, \mathcal{H}(r_i || p_i))\}$  (or e.g.  $\text{HMAC-}\mathcal{H}$ )
  - ▶ One has to check for every candidate  $p'$ , *for every user* if  $p'$  is the right password  $\leadsto$  no gain from multi-target

## But hash functions are too fast!

---

- ▶ If a password is “random enough”, salted hash is fine
- ▶ But most/some might not be that
- ▶ Assume that one:
  - ▶ Has  $2^{50}$  password candidates for a user
  - ▶ Can compute  $2^{23}$  hashes/core/second
  - ▶ Has 128 available cores
  - ▶  $\Rightarrow$  Only  $2^{20}$  seconds (< two weeks) to find  $p$  (that's not enough)
- ▶ One counter-measure: make hash functions *slower*
  - ▶ Not slow enough to hinder the user
  - ▶ Slow enough to make exhaustive search too costly

# First slow attempt: PBKDF2

---

- ▶ Instead of computing  $\mathcal{H}(r||p)$  once, iterate many times!
- ▶ Example: PBKDF2
  - ▶  $h \approx \bigoplus_{i=0}^c h_i$ ;  $h_i = \mathcal{H}(h_{i-1}||p)$ ;  $h_0 = r$
  - ▶ Choose the iteration count  $c$  to be “large enough”
  - ▶ Typically  $c \approx 1000$
- ▶ Say it takes 10ms to hash one password  $\Rightarrow$  35 years on 10 000 cores to try  $2^{50}$  candidates for one user
- ▶ One problem:
  - ▶ The user *needs* to hash on a regular core
  - ▶ An adversary may try hashes on fast dedicated circuits

A reasonable assumption:

- ▶ A PBKDF2 hash function can be computed  $2^{20}$  times faster than on a CPU core, using dedicated hardware with low amortized cost
- ▶ 10ms to hash one password on CPU  $\Rightarrow < 2^{-26}$ s on efficient hardware  $\Rightarrow < 2^{20}$  seconds on 10 machines to try  $2^{50}$  passwords

How to solve this?

- ▶ Cannot make the user wait one day to check a password
- ▶ So use hashing that's *slow everywhere*

# What's slow anyway?

---

An assumption: memory is similarly slow for everybody (CPU, GPU, FPGA, ASIC)

- ▶ So use a “memory-hard” hash function that needs a lot of memory to be computed
- ▶ A framework: the output must depend on “many” intermediate values, accessed many times  $\leadsto$  a (quadratic) tradeoff
  - ▶ Either store all intermediate values (costs memory)
  - ▶ Or recompute them as needed (costs time)
- ▶ Only increases memory consumption (not time) of hashing a password for a generic user
- ▶ Makes dedicated hardware not more efficient than regular CPU (hopefully)

## One memory-hard example: scrypt

---

Scrypt (Percival, 2009), the (very rough) idea:

- ▶ Use the password and salt to generate a large buffer
- ▶ Access the buffer many times in an unpredictable way to generate the output

A bit more precisely:

- 1  $h_i = \mathcal{H}(h_{i-1}); h_0 = r || p$ , for  $i$  up to  $n - 1$
- 2  $s_i = \mathcal{H}(s_{i-1} \oplus h_{s_{i-1} \bmod n})$ ,  $s_0 = \mathcal{H}(h_{n-1})$ , for  $i$  up to  $n$
- 3 Return  $s_n$

The intuitive tradeoff from two slides ago becomes:

- ▶ Either store all the  $h_i$ 's  $\leadsto$  time = memory  $\approx n$  calls to  $\mathcal{H}$ /accesses
- ▶ Either recompute  $h_{s_{i-1} \bmod n}$  once  $s_{i-1}$  is known  $\leadsto$  constant memory, time  $\approx n \times n/2$  calls to  $\mathcal{H}$
- ▶ Any combination in between (e.g. store one tenth of the  $h_i$ 's, regularly spaced)

$\Rightarrow$  Only a few MB of generated values might be enough to defeat special-purpose hardware

- ▶ One can in fact prove that the above tradeoff is roughly optimal (Alwen & al., 2016)

## An alternative approach: “Halting puzzles”

---

HKDF (Boyen, 2007) uses a memory-hard function with an (optionally) *unknown* iteration count

- 1 A user computes an iterated function on the password  $p$
  - 2 Interrupts the process when wanted; obtains a hash  $h$  of  $p$  and a verification string  $v$
  - 3 The hash and the iteration count can be retrieved from  $p$  and  $v$
- The user may tune the iteration count on its own to its requirements
  - Without that knowledge, an adversary is less efficient



# HKDF: How?

---

Preparation phase:

Input:  $p, r, t$

Output:  $h, v, r$

- 1  $z = \mathcal{H}(r||p)$
- 2 For  $i = 1, \dots, t \triangleleft t$  may be user-defined
- 3  $y_i = z$
- 4 For  $* = 1, \dots, q \triangleleft q$  controls the time/space ratio
- 5  $j = 1 + (z \bmod i)$
- 6  $z = \mathcal{H}(z||y_j)$
- 7 Return  $r; v = \mathcal{H}(y_1||z); h = \mathcal{H}(z||r)$

## HKDF: How? (bis)

---

Extraction phase:

Input:  $p, r, v$

Output:  $h$

- 1  $z = \mathcal{H}(r||p)$
- 2 For  $i = 1, \dots, \infty$
- 3      $y_i = z$
- 4     For  $* = 1, \dots, q$
- 5          $j = 1 + (z \bmod i)$
- 6          $z = \mathcal{H}(z||y_j)$
- 7         If  $(\mathcal{H}(y_1||z) = v)$  Then Break
- 8 Return  $h = \mathcal{H}(z||r)$

- ▶ Both functions use password-dependent memory accesses
- ▶ May leak information about the password (via side-channels)
- ▶ So (memory-hard) functions with password-independent accesses may sometimes be preferable
- ▶ For more on password hashing:  
<https://password-hashing.net/>

To finish: something a bit different

---



## To finish: something a bit different

---

It may be useful to have a hash function that:

- ▶ Is slow to execute (i.e. it is slow to compute  $y := \mathcal{H}(x)$  given  $x$ )
- ▶ Is fast to verify (i.e. it is fast to check that  $y = \mathcal{H}(x)$  given  $x$  and  $y$ )
- ▶  $\leadsto$  *Verifiable delay functions* (VDF)

An application:

- ▶ Collaborative random-number generation

## Randomness beacon

A *Randomness beacon* is a system that publishes (pseudo-)random numbers at regular interval

Example:

- ▶ <https://beacon.nist.gov/home>

Some applications:

- ▶ Remote random consensus ( “Shall we go to a pizzeria or a crêperie?” )
- ▶ (Faster) challenge generation in authentication protocols
- ▶ Lotteries
- ▶ Jury/assembly selection
- ▶ Random voting

# Collaborative beacons

---

One can distinguish:

- ▶ “Oracle” beacons (have to be trusted)
- ▶ “Collaborative” beacons (everyone can contribute)

A design strategy (Lenstra & Wesolowski, 2015):

- 1 Use a slow hash function with fast verification that takes wall time  $> \Delta$  to be computed (hopefully on the best platform)
  - 2 Gather public seeds from time  $t - \Delta$  to  $t$
  - 3 At time  $t$ , hash all collected seeds, then publish the hash
  - 4 Everyone can efficiently test the result and its dependence on the seeds
- ▶ An adversary does not have time to precompute a hash and insert a seed that biases the result

# A candidate slow hash function

---

Sloth: A slow hash function in a nutshell:

- ▶ If  $p \equiv 3 \pmod{4}$  is a (large) prime, if  $x \in \mathbb{F}_p^\times$  is a square mod  $p$ , the fastest known way to compute a square root of  $x$  is as  $x^{(p+1)/4}$
- ▶ Exactly one of  $x$  or  $-x$  is a square  $\Rightarrow$  one can map any number to a well-defined square root
- ▶ Computing a square root takes  $\approx \log(p)$  more time than “verifying” one

So (to make things more modular):

- ▶ Compute an iterative chain of square roots
- ▶ Interleaved with, say, block cipher applications to break the algebraic structure



## Some comments

---

- ▶ Sloth is not memory-hard, but CPUs are good at big-number arithmetic
  - ▶ Dedicated hardware may not be a threat
  - ▶ (Some password-hashing functions are based on the same assumption (Pornin, 2014))
- ▶ A Twitter-accessible beacon:  
`https://twitter.com/random\_zoo`
- ▶ The computation/verification gap in Sloth is not great asymptotically; better functions exist (cf. e.g. Wesolowski, 2019)