Security and Privacy

Password storage

6.10.2020

Ph. Oechslin

EPFL

Content

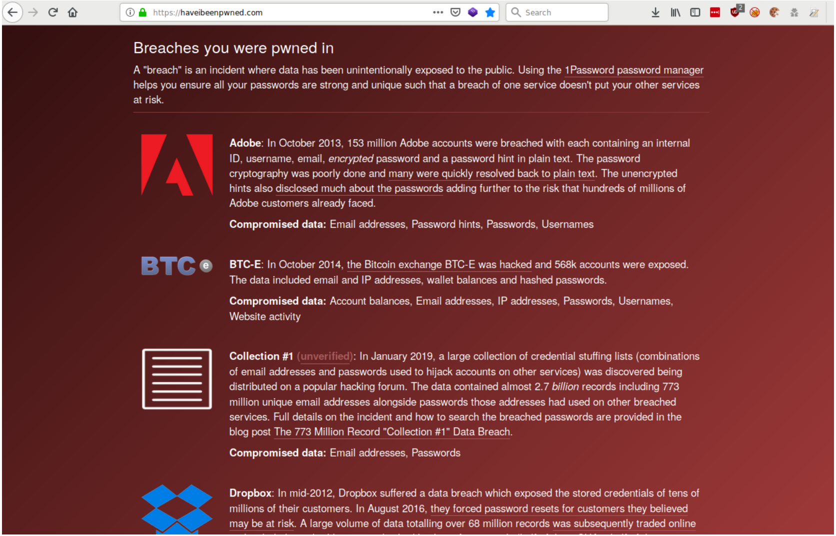
- Introduction
- The importance of salt
  - Time-memory trade-offs
  - Rainbow tables
- Storing hashes with salt and iterations
- Memory hard hash functions
  - Cracking Benchmark
- Conclusions

Introduction

Introduction

source: **Troy Hunt** January 2019

Introduction



https://haveibeenpwned.com

Search

Breaches you were pwned in

A "breach" is an incident where data has been unintentionally exposed to the public. Using the 1Password password manager helps you ensure all your passwords are strong and unique such that a breach of one service doesn't put your other services at risk.

Adobe: In October 2013, 153 million Adobe accounts were breached with each containing an internal ID, username, email, *encrypted* password and a password hint in plain text. The password cryptography was poorly done and many were quickly resolved back to plain text. The unencrypted hints also disclosed much about the passwords adding further to the risk that hundreds of millions of Adobe customers already faced.

Compromised data: Email addresses, Password hints, Passwords, Usernames

BTC-E: In October 2014, the Bitcoin exchange BTC-E was hacked and 568k accounts were exposed. The data included email and IP addresses, wallet balances and hashed passwords.

Compromised data: Account balances, Email addresses, IP addresses, Passwords, Usernames, Website activity

Collection #1 (unverified): In January 2019, a large collection of credential stuffing lists (combinations of email addresses and passwords used to hijack accounts on other services) was discovered being distributed on a popular hacking forum. The data contained almost 2.7 *billion* records including 773 million unique email addresses alongside passwords those addresses had used on other breached services. Full details on the incident and how to search the breached passwords are provided in the blog post The 773 Million Record "Collection #1" Data Breach.

Compromised data: Email addresses, Passwords

Dropbox: In mid-2012, Dropbox suffered a data breach which exposed the stored credentials of tens of millions of their customers. In August 2016, they forced password resets for customers they believed may be at risk. A large volume of data totalling over 68 million records was subsequently traded online

source: **haveibeenpwned.com**

Introduction

- 2,692,818,238 e-mail/password pairs published in Jan 2019
- Made up of thousands of different sources

➡ There must be something wrong with the way passwords are stored

Password storage

- Naive approach: cleartext
  - store passwords as clear text in database
  - 000webhost.com used to store passwords in clear-text
    - in 2015 ‘A hacker used an exploit in old PHP version of the website’ and stole 13 million passwords
  - you should never store passwords in cleartext
- Old school: hash the password
  - Microsoft stores Windows passwords as hashes (MD4)
  - Almost all passwords of length 8 can be recovered in under a minute

Password storage

- Classic way: use salt and iterations
  - Hugely slows down password cracking
  - Simple passwords can still be cracked on specialized hardware
- Modern way: use memory hard function
  - Cracking a password requires a decent amount of memory
  - Specialized hardware with many computing cores (e.g. graphic cards, FPGA) do not have enough memory to speed up cracking

The importance of salt

Microsoft does not salt their passwords.
Don’t be like Microsoft.

The importance of salt

- Some system store the passwords as a simple hash:

$$\mathrm{pw\_hash} = h(\mathrm{password})$$

- If passwords are hashed without any random information (salt) there are two major weaknesses:

1. Multiple hashes can be cracked at once
   - If you have a list of 1000 hashes and want to find out if any of the 1000 accounts has password ‘maison’;
     - calculate $h($‘maison’$)$
     - look-up which of the 1000 hashes matches
   - With a smart data structure, the look up is almost free,
     ➡ with a single hash operation you can try to crack 1000 passwords
   - Cracking n passwords does not require more hash operations than cracking a single password

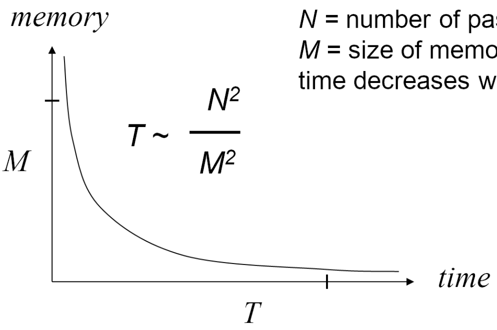
The importance of salt

2. Hashes can be calculated in advance

- Windows hashes have no salt:
  - ➡ every user on the earth with password 'maison' has the same hash 9c9948e891d31e09ad20b82b1796666a
- If we had enough memory, we could calculate all hashes in advance and store them in a great big table:
  - Eight mixed case characters plus digits: $62^8 = 2.2\,10^{14}$ passwords
  - We need 24 bytes to store a password and a hash
- Our table would be 5'240TB big!
- We can use time-memory trade-offs to
  - reduce the size of the table
  - while increasing the time needed to crack the passwords

Time-memory trade-offs

- In 1980, Martin Hellman invented a TMTO to inverse cryptographic functions
  - when you double the amount of memory used, it is four times faster to invert the function



$$T \sim \frac{N^2}{M^2}$$

N = number of passwords
M = size of memory
time decreases with square of memory

- Rainbow Tables are an optimization of this TMTO from 2003 (by your's truly)

Time-memory trade-offs

- Basic idea: we organize hashes in chains
- We agree on a set of passwords to crack (e.g. 8 alphanum characters)
- We create a reduction function r: it takes a hash as input and produces a password from our set
- We can now build chains:

$$p_1 \xrightarrow{hash} h_1 \xrightarrow{reduce} p_7 \xrightarrow{hash} h_7 \xrightarrow{reduce} p_8 \xrightarrow{hash} h_8 \xrightarrow{reduce} p_3 \xrightarrow{hash} h_3$$

- We only keep the first and the last element of each chain
  - this is where we save memory
  - we pay for this with more time to crack the passwords

Time-memory trade-offs

- Let's build a table:
  - we create four chains and only store the first and last elements

$$
\begin{array}{ccccccccccccccccc}
\color{darkred}{p_1} & \xrightarrow{h} & h_1 & \xrightarrow{r} & p_7 & \xrightarrow{h} & h_7 & \xrightarrow{r} & p_8 & \xrightarrow{h} & h_8 & \xrightarrow{r} & p_4 & \xrightarrow{h} & \color{darkred}{h_4} \\
\color{darkred}{p_2} & \xrightarrow{h} & h_2 & \xrightarrow{r} & p_9 & \xrightarrow{h} & h_9 & \xrightarrow{r} & p_0 & \xrightarrow{h} & h_0 & \xrightarrow{r} & p_1 & \xrightarrow{h} & \color{darkred}{h_1} \\
\color{darkred}{p_3} & \xrightarrow{h} & h_3 & \xrightarrow{r} & p_5 & \xrightarrow{h} & h_5 & \xrightarrow{r} & p_7 & \xrightarrow{h} & h_7 & \xrightarrow{r} & p_8 & \xrightarrow{h} & \color{darkred}{h_8} \\
\color{darkred}{p_4} & \xrightarrow{h} & h_4 & \xrightarrow{r} & p_6 & \xrightarrow{h} & h_6 & \xrightarrow{r} & p_3 & \xrightarrow{h} & h_3 & \xrightarrow{r} & p_5 & \xrightarrow{h} & \color{darkred}{h_5}
\end{array}
$$

- typically, chains would contain tens of thousands of hashes and passwords

Time-memory trade-offs

- Let's try to crack h_6.
  - remember, we only store the first and the last element of each line

$$
\begin{array}{ccccccccccccccc}
p_1 & \xrightarrow{h} & h_1 & \xrightarrow{r} & p_7 & \xrightarrow{h} & h_7 & \xrightarrow{r} & p_8 & \xrightarrow{h} & h_8 & \xrightarrow{r} & p_4 & \xrightarrow{h} & h_4 \\
p_2 & \xrightarrow{h} & h_2 & \xrightarrow{r} & p_9 & \xrightarrow{h} & h_9 & \xrightarrow{r} & p_0 & \xrightarrow{h} & h_0 & \xrightarrow{r} & p_1 & \xrightarrow{h} & h_1 \\
p_3 & \xrightarrow{h} & h_3 & \xrightarrow{r} & p_5 & \xrightarrow{h} & h_5 & \xrightarrow{r} & p_7 & \xrightarrow{h} & h_7 & \xrightarrow{r} & p_8 & \xrightarrow{h} & h_8 \\
p_4 & \xrightarrow{h} & h_4 & \xrightarrow{r} & p_6 & \xrightarrow{h} & h_6 & \xrightarrow{r} & p_3 & \xrightarrow{h} & h_3 & \xrightarrow{r} & p_5 & \xrightarrow{h} & h_5
\end{array}
$$

  - we check if h_6 is a known end of chain: it is not
  - we reduce and hash: $h_6 \xrightarrow{r} p_3 \xrightarrow{h} h_3$
    still not a known end of chain
  - we reduce and hash once more: $h_6 \xrightarrow{r} p_3 \xrightarrow{h} h_3 \xrightarrow{r} p_5 \xrightarrow{h} h_5$
    yes! we found an end of chain

Time-memory trade-offs

- We know that the password of h_6 in the the last chain
- We have stored the first element of the chain
- We can thus reconstruct the chain up to p_6

$$
\begin{array}{l}
p_1 \xrightarrow{h} h_1 \xrightarrow{r} p_7 \xrightarrow{h} h_7 \xrightarrow{r} p_8 \xrightarrow{h} h_8 \xrightarrow{r} p_4 \xrightarrow{h} h_4 \\
p_2 \xrightarrow{h} h_2 \xrightarrow{r} p_9 \xrightarrow{h} h_9 \xrightarrow{r} p_0 \xrightarrow{h} h_0 \xrightarrow{r} p_1 \xrightarrow{h} h_1 \\
p_3 \xrightarrow{h} h_3 \xrightarrow{r} p_5 \xrightarrow{h} h_5 \xrightarrow{r} p_7 \xrightarrow{h} h_7 \xrightarrow{r} p_8 \xrightarrow{h} h_8 \\
p_4 \xrightarrow{h} h_4 \xrightarrow{r} \boldsymbol{p_6} \xrightarrow{h} h_6 \xrightarrow{r} p_3 \xrightarrow{h} h_3 \xrightarrow{r} p_5 \xrightarrow{h} h_5
\end{array}
$$

- we have cracked the password!
- by storing only the start and end of 4 chains we can crack any of the 10 passwords contained in the chains

Time-memory trade-offs

- Hellman's original trade-off becomes inefficient when there are too many chains in a single table
  - For each collision of the reduction function, we end up with two identical chains

$$
\begin{array}{cccccccccccccccc}
p_1 & \xrightarrow{h} & h_1 & \xrightarrow{r} & p_7 & \xrightarrow{h} & h_7 & \xrightarrow{r} & p_8 & \xrightarrow{h} & h_8 & \xrightarrow{r} & p_4 & \xrightarrow{h} & h_4 \\
p_2 & \xrightarrow{h} & h_2 & \xrightarrow{r} & p_9 & \xrightarrow{h} & h_9 & \xrightarrow{r} & p_0 & \xrightarrow{h} & h_0 & \xrightarrow{r} & p_1 & \xrightarrow{h} & h_1 \\
p_3 & \xrightarrow{h} & h_3 & \xrightarrow{r} & p_5 & \xrightarrow{h} & h_5 & \xrightarrow{r} & p_7 & \xrightarrow{h} & h_7 & \xrightarrow{r} & p_8 & \xrightarrow{h} & h_8 \\
p_4 & \xrightarrow{h} & h_4 & \xrightarrow{r} & p_6 & \xrightarrow{h} & h_6 & \xrightarrow{r} & p_3 & \xrightarrow{h} & h_3 & \xrightarrow{r} & p_5 & \xrightarrow{h} & h_5
\end{array}
$$

- Hellman's solution was to use many small tables with different reduction function each
  - you need to search for the password in each table separately

Rainbow tables

- Rainbow tables solve the collision problem by using a different reduction function in each row.
  - This allows building much large tables and makes them much more efficient
    - less hash operations, much less memory look-ups

$$
\begin{array}{ccccccccccccc}
p_1 & \xrightarrow{h} & h_1 & \xrightarrow{\mathbf{r_1}} & p_7 & \xrightarrow{h} & h_7 & \xrightarrow{\mathbf{r_2}} & p_2 & \xrightarrow{h} & h_2 & \xrightarrow{\mathbf{r_3}} & p_7 & \xrightarrow{h} & h_7 \\
p_2 & \xrightarrow{h} & h_2 & \xrightarrow{\mathbf{r_1}} & p_9 & \xrightarrow{h} & h_9 & \xrightarrow{\mathbf{r_2}} & p_4 & \xrightarrow{h} & h_4 & \xrightarrow{\mathbf{r_3}} & p_3 & \xrightarrow{h} & h_3 \\
p_3 & \xrightarrow{h} & h_3 & \xrightarrow{\mathbf{r_1}} & p_5 & \xrightarrow{h} & h_5 & \xrightarrow{\mathbf{r_2}} & p_3 & \xrightarrow{h} & h_3 & \xrightarrow{\mathbf{r_3}} & p_5 & \xrightarrow{h} & h_5 \\
p_4 & \xrightarrow{h} & h_4 & \xrightarrow{\mathbf{r_1}} & p_6 & \xrightarrow{h} & h_6 & \xrightarrow{\mathbf{r_2}} & p_5 & \xrightarrow{h} & h_5 & \xrightarrow{\mathbf{r_3}} & p_1 & \xrightarrow{h} & h_1
\end{array}
$$

Searching in a rainbow table

- lets try to crack h_6:
  - $$\begin{array}{l}
p_1 \xrightarrow{h} h_1 \xrightarrow{\mathbf{r_1}} p_7 \xrightarrow{h} h_7 \xrightarrow{\mathbf{r_2}} p_2 \xrightarrow{h} h_2 \xrightarrow{\mathbf{r_3}} p_7 \xrightarrow{h} h_7 \\
p_2 \xrightarrow{h} h_2 \xrightarrow{\mathbf{r_1}} p_9 \xrightarrow{h} h_9 \xrightarrow{\mathbf{r_2}} p_4 \xrightarrow{h} h_4 \xrightarrow{\mathbf{r_3}} p_3 \xrightarrow{h} h_3 \\
p_3 \xrightarrow{h} h_3 \xrightarrow{\mathbf{r_1}} p_5 \xrightarrow{h} h_5 \xrightarrow{\mathbf{r_2}} p_3 \xrightarrow{h} h_3 \xrightarrow{\mathbf{r_3}} p_5 \xrightarrow{h} h_5 \\
p_4 \xrightarrow{h} h_4 \xrightarrow{\mathbf{r_1}} p_6 \xrightarrow{h} h_6 \xrightarrow{\mathbf{r_2}} p_5 \xrightarrow{h} h_5 \xrightarrow{\mathbf{r_3}} p_1 \xrightarrow{h} h_1
\end{array}$$
  - we check if h_6 is an end of chain, it is not

$$h_6$$

  - we reduce and hash, still nothing

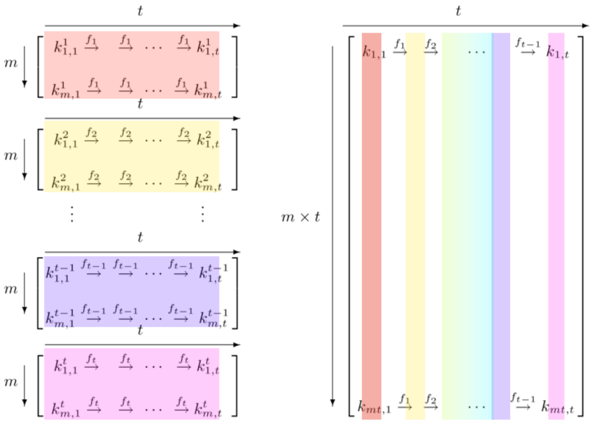
$$h_6 \xrightarrow{\mathbf{r_3}} p_2 \xrightarrow{h} h_2$$

  - we reduce and hash, twice: bingo!

$$h_6 \xrightarrow{\mathbf{r_2}} p_5 \xrightarrow{h} h_5 \xrightarrow{\mathbf{r_3}} p_1 \xrightarrow{h} h_1$$

Performance

- compare t Hellman tables with m chains of length t against a single rainbow table of $m \times t$ chains of length t



source: **wikipedia**

Performance

- If you search through all columns of all tables:
  - Hellman: t^2 memory look-ups, t^2 hash operations
  - Rainbow: t memory look-ups, $\frac{1}{2}t^2$ hash operations ($1 + 2 + .. + n$)
  - t times fewer look-ups, 2 times fewer hash operations
- If you search through half of the columns
  - t times fewer look-ups, 4 times fewer hash operations

➡ the higher the success rate, the better the rainbow table

- A 2.5TB rainbow table can invert hashes of passwords made of 8 mixed cases letters, digits and 33 special characters (2^{52}) in less than a minute on a two-processor server (**demo**)
  - equivalent to about 50 tera-hashes per second, for a single password
  - NVIDIA RTX 3080: 93 giga-hashes per second (**source**)

Storing hashes with salt and iterations

The classical way

Using salt

- Adding a random value (salt) to the hash function prevents the two issues we saw
  - you can not crack multiple hashes with a single hash calculation
  - you can not calculate the hashes in advance

- Because each hash as a different, random salt

$$\text{salt} = random()$$

$$\text{pw\_hash} = h(\text{password}, \text{salt})$$

- We need to store both the hash and the salt in the database
  - when a user logs in, we use the salt to generate a hash and compare it to the stored hash

Quiz !

- What cryptographic primitive can we use to combine a salt with a hash ?

➡

Salt is not enough

- Cryptographic hash functions are designed to be very fast and simple to implement.
- A modern graphic card can typically calculate hundreds of billions of hashes per second.
- A simple way of slowing the attacker down is to apply the hash function multiple times.
- If you require 5000 iterations for creating the password hash
  - login will take 5000 times longer (e.g. 0.05s instead of 0.00001s)
  - cracking will be 5000 times slower (e.g. 2 years instead of 4 hours)

Salt and iterations: standards

- There is an official standard for using salt and iterations in hash functions
- The current version is Password Based Key Derivation Function 2 (PBKDF2, **RFC 8018**)
- Used for example in
  - Wi-Fi WPA
  - MacOS user password hashes
  - Linux disk encryption (LUKS)

Salt and iterations in modern OSes

- Linux
  - default Linux uses 48 bits of salt and 5000 iterations of SHA512 to create a password hash.
  - The salt and hash are stored in `/etc/shadow`:

```
philippe:$6$Xi5sBXT5$CBBcKyahJMymJpKHfVQhY273n2cA8MmSYjC19W5cn1rIK
  Fvq4beaHFszeU5nQ.XfJHXoacWwjJ91q5Ic/.27x0:17537:0:99999:7:::
```

    `$6$` is the type of hash (sha512), `Xi5sBXT5` is the salt, `CBB..7x0` the hash

  - The number of iterations is specified in `/etc/login.defs`
- MacOS
  - 256 bits of salt, SHA512, the number of iterations is adjusted to take 0.1 seconds on login.
    - the number of iteration is stored with the salt and the hash in `/var/db/dslocal/nodes/Default/users/username.plist`

Memory hard hash functions

The modern way of hashing passwords

Memory hard functions

- Iterating a cryptographic hash function is not the most efficient way to slow down an attacker.

- Specialized hardware (graphic cards, FPGAs) can easily compute thousands of hashes in parallel.
  - e.g. it takes less than 50k transistors to implement SHA512

- Modern password hash functions require a certain amount of memory (e.g. 16MB).
  - For a single login operation, 16MB are easily available
  - Graphic cards or FPGAs would need gigabytes of internal memory to parallelize thousands of hashes
  - e.g. it takes millions of transistors to store 16MB of data

Memory hard functions

- The functions run through many steps
- Intermediate results are stored in memory
- Each step depends on results from previous steps
- If you do not have enough memory you can still calculate the result
  - but you have to recalculate intermediate values again and again
- Typical memory hard hash functions can be parametrized:
  - chose the amount of memory needed
  - chose the number of steps to calculate

Memory hard password hash functions

- Scrypt
  - Invented in 2012 by Colin Percival and standardized in 2016 (**RFC 7914**).
    - typical configuration uses about 16MB of memory and less than 100ms of CPU time
    - parameters can be adapted to reflect capabilities of current hardware
- Argon2
  - Argon2 (by A Biryukov et al.) was selected 2015 as winner of the password hashing competition organized by JP Aumasson and other cryptographers

Cracking Benchmark

- Here is a benchmark of the Hashcat password cracker on a GeForce RTX 3080 graphics card

```
NTLM                           93430.6 MH/s  (windows, no iterations)
sha512crypt $6$                  373.2 kH/s  (linux, 5000 iterations)
OSX v10.8+ (PBKDF2-SHA512)      1019.2 kH/s  (OSX,   1023 iterations)
Cisco-IOS $9$ (scrypt)            42.4 kH/s  (N = 16384, r = 1, p =1)
```

- When implementing password storage
  - Always use salt and make the hash function slow
  - Use Scrypt or Argon2 if available
  - If not, use PBKDF2

Conclusions

Conclusions

- NEVER store passwords in clear text
- Always use salt when storing passwords. If not:
  - multiple hashes can be cracked at once
  - hashes can be calculated in advance
    - rainbow tables are a very efficient way to store this information
- Salt is not enough, we must slow down the hash function
  - Iterations are a good way to make the hash slower
    - PBKDF2 is the standard way of doing it
  - Memory hard functions are even better
    - They are more expensive to parallelize
      - GPUs and FPGAs have little internal memory
    - Examples: Scrypt, Argon2