Passwords

https://shop.spectator.co.uk/wp-content/uploads/2015/03/open-sesame.jpg
What’s in a password?

- 12345
- 123456789
- password
- adobe123
- 12345678
- qwerty
- 1234567
- 111111
- photoshop

- 123123
- 12344567890
- 000000
- abc123
- macromedia
- azerty
- iloveyou
- aaaaaa
- 654321
What’s in a password?

- 12345
- 123456789
- password
- adobe123
- 12345678
- qwerty
- 1234567
- 111111
- photoshop

- 123123
- 12344567890
- 000000
- abc123
- macromedia
- azerty
- iloveyou
- aaaaaa
- aaaaaaa
- 654321

These passwords were used by over 4 million accounts. How can we fix this?
What are passwords?

Secret token used to prove that you’re you
Key property: only need to be able to verify that it’s correct
Password Verification

Client

Username, Password

Authentication Server

Success / Failure
Password Verification

Client → Authentication Server
Username, Password
Success / Failure

Where could we attack?
Password Verification

Client → Username, Password → Authentication Server

Client device → Communication (usually network) → Authentication Server

Success / Failure
Password Verification

Client → Username, Password → Authentication Server

Client device → Communication (usually network) → Authentication Server

Success / Failure
How should we store the passwords on the server?
Threat Model Assumptions

Attacker’s primary method is exfiltration, and Attacker can’t modify code

- These assumptions model the typical case and
- If the attacker can make more important changes, the incursion is probably complete
How should we store the passwords on the server?
Attempt #1 - Plaintext

- Store passwords in plaintext

Client

user, $u_2$; password, $p$

Success / Failure

Server

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_2$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>$u_3$</td>
<td>$p_3$</td>
</tr>
</tbody>
</table>

$p == p_2$?
Attempt #1 - Plaintext

- Store passwords in plaintext
- What could go wrong?
Attempt #1 - Plaintext

- Store passwords in plaintext
- What could go wrong?
  - If database is stolen, so are passwords!
  - Admins have access to passwords.

Attempt #2 - Encryption

- Encrypt the passwords
- Decrypt and compare on login

Client → Server

- User, $u_2$; password, $p$
- Encryption key
- Encryption of $p_1$ with $k$
- Decryption of $p_2'$ with $k$

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$p_1' = E_k(p_1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_2$</td>
<td>$p_2' = E_k(p_2)$</td>
</tr>
<tr>
<td>$u_3$</td>
<td>$p_3' = E_k(p_3)$</td>
</tr>
<tr>
<td>$p$</td>
<td>$D_k(p_2)$?</td>
</tr>
</tbody>
</table>
Attempt #2 - Encryption

- Encrypt the passwords
- Decrypt and compare on login
- What advantages does this scheme have?
Attempt #2 - Encryption

- Encrypt the passwords
- Decrypt and compare on login
- What advantages does this scheme have?
  - If database is stolen, passwords can't be read
  - Only administrators with the encryption key can read the passwords
Attempt #2 - Encryption

- Encrypt the passwords
- Decrypt and compare on login
- What advantages does this scheme have?
  - If database is stolen, passwords can’t be read
  - Only administrators with the encryption key can read the passwords
- What could go wrong?
Attempt #2 - Encryption

● Encrypt the passwords
● Decrypt and compare on login
● What advantages does this scheme have?
  ○ If database is stolen, passwords can’t be read
  ○ Only administrators with the encryption key can read the passwords

● What could go wrong?
  ○ If the database is stolen, what is to keep the key from being stolen?
  ○ Anyone with the key (admins) can view passwords

● ex. Adobe (2013)
Attempt #2 - Encryption

```
4464---|--|xxx@yahoo.com|--g2B6PhWEH366cdBSCql/UQ==|--try: qwerty123|--
4465---|--|xxxxx@jcom.home.ne.jp|--Eh5tLomK+N+82csoVwU9bw==|--?????|--
4466---|--|xx@hotmail.com|--ahw2b2BELzgRTWYvQGn+kw==|--quiero a...|--
4467---|--|xxx@yahoo.com|--leMTcMPEPcjioxG6CatHBw==|--
4468---|--|username|xxxxx@adobe.com|--2GtbVrmsERzioxG6CatHBw==|--
4469---|--|xxx@yahoo.com|--4LSlo772tH4|--rugby|--
4470---|--|xxx@hotmail.com|--WXGzX56zRXnioxG6CatHBw==|--
4471---|--|xxxx@yahoo.com|--x3eI/bgfnrioxxG6CatHBw==|--myspace|--
4471---|--|xxx@hotmail.com|--kbyi9I8wDrrioxxG6CatHBw==|--regular|--
```
Attempt #2 - Encryption

Even without the key, it’s still bad. Why?
Even without the key, it’s still bad. Why?

*Identical passwords produce identical ciphertexts*

If you know one password, you know all with the same ciphertext

Frequency analysis (*0.5% of users use password*)

Password hints (*“numbers 123456”*)
Even without the key, it’s still bad. Why?

Some encryption functions produce variable length output that depends on the input.
# Attempt #2 - Encryption

<table>
<thead>
<tr>
<th>Password data (hex)</th>
<th>Password hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b4c27d8f75cc41a</td>
<td>-&gt; Same old, same old</td>
</tr>
<tr>
<td>e826ef87cc7a3029</td>
<td>-&gt; You'll never guess</td>
</tr>
<tr>
<td>e2a311ba09ab4707</td>
<td>-&gt; Dog + digit</td>
</tr>
<tr>
<td>0842cbb7edf3e343</td>
<td>-&gt; Dog</td>
</tr>
<tr>
<td>e2a311ba09ab4707</td>
<td>-&gt; Virtuously long</td>
</tr>
<tr>
<td>92663700893c3f27</td>
<td>-&gt; Geburtestag</td>
</tr>
<tr>
<td>a667d747891a8255</td>
<td></td>
</tr>
<tr>
<td>88fc540356d561ec</td>
<td></td>
</tr>
<tr>
<td>fb0a9047a5dd5ef8</td>
<td></td>
</tr>
<tr>
<td>f3c512b0e38a5392</td>
<td></td>
</tr>
<tr>
<td>a3f492fbd917f632</td>
<td></td>
</tr>
<tr>
<td>92bb535704f0ae7f</td>
<td></td>
</tr>
</tbody>
</table>
Attempt #3 - Hashing

Password → Hash Function → Hash

Example:

1337p4Ss → SHA2 → a487cb0eeb4a484a269c703bce7f8c46b53d4860267a24900ae7ceb577315eb1
Recall cryptographic hashing:
  - Variable length input, fixed length “random” output
  - One-way
    - Given hash $x$, hard to find $p$ such that $H(p) = x$
  - Weak collision resistance
    - Given input $p$, hard to find $q$ such that $H(p) = H(q)$
  - Strong collision resistance
    - Hard to find distinct $p, q$ such that $H(p) = H(q)$
Attempt #3 - Hashing

- Hash the password, store the hash
- Hash the given password and compare

Server

<table>
<thead>
<tr>
<th>User</th>
<th>Hashed Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>$p_1' = H(p_1)$</td>
</tr>
<tr>
<td>$u_2$</td>
<td>$p_2' = H(p_2)$</td>
</tr>
<tr>
<td>$u_3$</td>
<td>$p_3' = H(p_3)$</td>
</tr>
</tbody>
</table>

$H(p) == p_2'$?
1. Hash the password, store the hash
2. Hash the given password and compare
3. What advantages does this scheme have?
Attempt #3 - Hashing

1. Hash the password, store the hash
2. Hash the given password and compare
3. What advantages does this scheme have?

If database is stolen, hashes need to be cracked
Cracking is hard, so attackers get fewer passwords
Attempt #3 - Hashing

What could go wrong?
What could go wrong?

- *Identical passwords produce identical hashes*
- Once you’ve cracked a given hash, you can trivially crack it every time you see the same hash again
- Frequency analysis
- Precompute massive tables for popular hash functions
  - These are called *rainbow tables*
- Common passwords are *very common*
  - Even a small table cracks most passwords
Attempt #4 - Salting

• What is salt?
Salt

Common salt is a mineral substance composed primarily of sodium chloride (NaCl), a chemical compound belonging to the larger class of ionic salts; salt in its natural form as a crystalline mineral is known as rock salt or halite. Salt is present in vast quantities in the sea where it is the main mineral constituent, with the open ocean having about 35 grams (1.2 oz) of solids per litre, a salinity of 3.5%. Salt is essential for animal life, and saltiness is one of the basic human tastes. The tissues of animals contain larger quantities of salt than do plant tissues; therefore the typical diets of nomads who subsist on their flocks and herds require little or no added salt, whereas cereal-based diets require supplementation. Salt is one of the oldest and most ubiquitous of food seasonings, and salting is an important method of food preservation.

source: https://en.wikipedia.org/wiki/Salt

As a CIA officer, Evelyn Salt (Jolie) swore an oath to duty, honor and country. Her loyalty will be tested when a defector accuses her of being a Russian spy. Salt goes on the run, using all her skills and years of experience as a covert operative to elude capture. Salt's efforts to prove her innocence only serve to cast doubt on her motives, as the hunt to uncover the truth behind her identity continues and the question remains: "Who Is Salt?"

Rating: SHA-13 (for intense sequences of violence and cryptography)

Attempt #4 - Salting

Password
Salt (Random)

Hash Function

Hash

SHA2

1337p4Ss
fA4dY5

35e89a9469522038161a3c173815d8e7d
6ed2c6957eb5d522b8e0942cf93ea72
Attempt #4 - Salting

- Hashing the same password with different salt will produce different hashes.

- **SHA2 fa80328ef40ecbf22943747d8fe63e3b57bc9ee094db754ca74e034875deb1d**
- **SHA2 2674d6e9c01f5ea3235cafccac433a30a9de88ddfa0c2e044b53daa63c8afdd**
Attempt #4 - Salting

- Hash the password, store the hash
- Hash the given password and compare

$$u_1 \quad p_1' = H(p_1, s)$$

$$u_2 \quad p_2' = H(p_2, s)$$

$$u_3 \quad p_3' = H(p_3, s)$$

$$H(p, s) == p_2' ?$$
Attempt #4 - Salting

- Hash the password, store the hash
- Hash the given password and compare
- What advantages does this scheme provide?
Attempt #4 - Salting

- Hash the password, store the hash
- Hash the given password and compare
- What advantages does this scheme provide?
  - In order to precompute, need password *and salt*
  - Since salts are random, guessing a salt is useless
  - Even if salt is known, computation must be redone for every site
Attempt #4 - Salting

- What could go wrong?
Attempt #4 - Salting

- What could go wrong?
  - Identical passwords and identical salts produce identical hashes
  - Frequency analysis
  - If you crack one password, you crack them all
  - For big sites, precomputation is worth it
Attempt #4 - Salting

- How could we make the analysis even harder?
Attempt #5 - Per-User Salting

- Generate a salt, hash the password, store salt and hash
- Hash the given password with the user’s salt and compare

Client

Server

<table>
<thead>
<tr>
<th>User</th>
<th>Salt</th>
<th>Hash with Salt</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>$s_1$</td>
<td>$p_1' = H(p_1, s_1)$</td>
</tr>
<tr>
<td>$u_2$</td>
<td>$s_2$</td>
<td>$p_2' = H(p_2, s_2)$</td>
</tr>
<tr>
<td>$u_3$</td>
<td>$s_3$</td>
<td>$p_3' = H(p_3, s_3)$</td>
</tr>
</tbody>
</table>

$H(p, s_2) == p_2'$?
Attempt #5 - Per-User Salting

- Generate a salt, hash the password, store the hash
- Hash the given password with the user’s salt and compare
- What are the advantages of this scheme?
Attempt #5 - Per-User Salting

- Generate a salt, hash the password, store the hash
- Hash the given password with the user’s salt and compare
- What are the advantages of this scheme?
  - Since every user has a different salt, *identical passwords will not have identical hashes*
  - No frequency analysis
  - No using known passwords to crack other passwords
  - No precomputation - much harder to crack
Work Factor

- Measure of “work” needed **per cracked password**
- Intuitively, higher work factor means greater security
- Work factor for plaintext model is nominally 1.
- For non-salted hashes, construction of rainbow table requires a large but fixed pre-computation. Work per cracked password decreases with number of crackings.
Work Factor

• Measure of “work” needed per cracked password
• Work factor for plaintext model is nominally 1.
• For non-salted hashes, construction of rainbow table requires a large but fixed pre-computation.
• For individually salted passwords, work is maximal.
Slow Hashes

Work factor increases if hashing is slow

ex., bcrypt, PBKDF2 are designed to be slow

Both feature a work factor parameter
This work factor tunes how many times the hash is computed ($H(p)$ vs $H(H(p))$, etc)
Password Cracking Methods
Brute Force

1. Brute force
   • Try all passwords in a given space
   • Eventually succeeds given enough time and CPU power
2. Dictionary

- Precompute hashes of a set of likely passwords
- Store (hash, password) pairs sorted by hash
- Fast look up for password given the hash
- Requires large storage and preprocessing time
Rainbow Tables

3. Rainbow table

• Table of previously cracked passwords with their hashes
• More storage, shorter cracking time
Password Cracking Tradeoff

Time

Brute force
Rainbow table
Dictionary

Storage
Complexity?

- 10 digits
- Lower case characters
- UPPER and lower case
- Special characters
- Standard keyboard characters
- All 7-bit ASCII characters
Size of Password Space

6 character password (can **only** use lower case letters, no numbers):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
10 character password (can **only** use lower case letters, no numbers):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Index 0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td># of choices</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Size of Password Space

10 character password (can only use lower case letters, no numbers):

<table>
<thead>
<tr>
<th>Char</th>
<th>b</th>
<th>h</th>
<th>?</th>
<th>?</th>
<th>?</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td># of choices</td>
<td>26</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Size of Password Space**

10 character password (can **only** use lower case letters, no numbers):

<table>
<thead>
<tr>
<th>Char</th>
<th>b</th>
<th>h</th>
<th>?</th>
<th>?</th>
<th>?</th>
<th>?</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td># of choices</td>
<td>26</td>
<td>26</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Result: $26 \times 26 \ldots \times 26$ or $26^6$ possible passwords!
How many possible passwords?
• Digits (10): $10^6$
• UPPPER and lower case (52): $10^{52}$
• Special characters: &, %, $, @, “, |, ^, <, ... (32): $32^6$
• Standard keyboard characters (94): $94^6$
• All 7-bit ASCII characters (128): $128^6$
### Password Length

Assume a standard keyboard with 94 characters.

<table>
<thead>
<tr>
<th>Password length</th>
<th>Number of passwords</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$94^5 = 7,339,040,224$</td>
</tr>
<tr>
<td>6</td>
<td>$94^6 = 689,869,781,056$</td>
</tr>
<tr>
<td>7</td>
<td>$94^7 = 64,847,759,419,264$</td>
</tr>
<tr>
<td>8</td>
<td>$94^8 = 6,095,689,385,410,816$</td>
</tr>
<tr>
<td>9</td>
<td>$94^9 = 572,994,802,228,616,704$</td>
</tr>
</tbody>
</table>
Texas Attorney General

Rules

1. The password must be **exactly** 8 characters long.
2. It must contain **at least** one letter, one number, and one of the following special characters.
   a. The **only** special characters allowed are: @ # $
   b. A special character must **not** be located in the first or last position.
3. Two of the same characters sitting next to each other are considered to be a “set.” No “sets” are allowed. **Example:** rr, tt
4. Avoid using names, such as your name, user ID, or the name of your company or employer.
5. Other words that cannot be used are Texas, child, and the months of the year.
6. A new password cannot be too similar to the previous password.
   a. **Example:** previous password - abc#1234; unacceptable new password - acb$1243
   b. Characters in the first, second, and third positions cannot be identical. (abc****)
   c. Characters in the second, third, and fourth positions cannot be identical. (*bc#****)
   d. Characters in the sixth, seventh, and eighth positions cannot be identical. (*****234)
7. A password can be changed voluntarily (no Help Desk assistance needed) once in a 15-day period. If needed, the Help Desk can reset the password at any time.
8. The previous 8 passwords cannot be reused.
Intelligent Guessing Methods

Try the top $n$ most common passwords
Dictionary of words, names, etc
Syntax model
  e.g., “dictionary word with some letters replaced by numbers” - elitenoob, eliten00b, 31it3n00b
Markov chain model
Intelligent Guessing

For any scheme that involves guessing, work factor is reduced by *guessing intelligently*.  
Key insight: *not all passwords are equally likely* 
Idea: try most likely passwords first 
In reality, it doesn’t take 9 years to brute force most passwords
How Unequal?

From a study of the Adobe breach

- password - 0.5%
- a password in the top 10 - 1.6%
- a password in the top 100 - 4.4%
- a password in the top 1,000 - 13.2%
- a password in the top 10,000 - 30%

If you’re interested, check out these master lists of passwords on Daniel Meissler’s github
Ashley Madison

• In 2015, 11 million of the 36 million passwords on dating website Ashley Madison were cracked
• How did this happen? And why weren’t all 36 million cracked?
Ashley Madison

• Security expert Dean Pierce attempted to brute force the stolen hashes, using master list of plaintext passwords RockYou
• Didn’t have much luck – cracked less than 1%
Ashley Madison

• Security expert Dean Pierce attempted to brute force the stolen hashes, using master list of plaintext passwords RockYou
• Didn’t have much luck – cracked less than 1%
• So how did the hackers do it?
Ashley Madison

- Variable $loginkey
- Prior to an update in 2012, this variable had actually stored the user’s plaintext password
Ashley Madison

- Variable $loginkey
- Prior to an update in 2012, this variable had actually stored the user’s plaintext password
- The update involved implementing the bcrypt hash function, but stored account info was not updated
- Thus, accounts made after 2012 were still stored in the clear
Hashcat Demo

- **Hashcat** is a CPU-based password recovery tool