User Authentication

Kostas Papagiannopoulos
University of Amsterdam
k.papagiannopoulos@uva.nl

Contents

Introduction

Introduction

Authentication. Determining whether a user (or other entity) should be allowed access to a particular system or resource.

Introduction

Authentication. Determining whether a user (or other entity) should be allowed access to a particular system or resource.

- Authentication implies a one-to-one test: the user asserts an identity and the server determines if that is true or false
- Human beings are very good at authenticating to each other by recognizing face, voice etc.
- The problem becomes more complex when a human must interact and authenticate with a machine

Introduction

A human can be authenticated to a machine using various methods:

- Use something he knows
  e.g. remember a password or PIN code to login to his email account
- Use something he has
  e.g. insert the smartcard to the bank ATM, open the car door with an RFID tag
- Use something he is
  e.g. scan his fingerprints or face to unlock a smartphone

Passwords

Passwords

- Passwords is the most common way humans use to authenticate to a system
- Passwords take various forms depending on the application context

  e.g. the email password consists of many alphanumeric characters, possibly together with special characters

  e.g. the PIN code consists of 4 digits, each from 0 until 9

  e.g. cryptocurrency wallets ask you to setup a lengthy passphrase for backup

Passwords

- Passwords is the most common way humans use to authenticate to a system
- Passwords take various forms depending on the application context

  e.g. the email password consists of many alphanumeric characters, possibly together with special characters

  e.g. the PIN code consists of 4 digits, each from 0 until 9

  e.g. cryptocurrency wallets ask you to setup a lengthy passphrase for backup
- In theory a user could use a random key as their password
- This can be hard to remember but still a valid option
- Passwords are easy to transfer, which is good and bad

Passwords

Comparing cryptographic keys to passwords

- Let a randomly chosen 64-bit key. How many are the possible keys?

$$\text{number of keys } = 2^{64}$$

Passwords

Comparing cryptographic keys to passwords

- Let a randomly chosen 64-bit key. How many are the possible keys?

$$\text{number of keys } = 2^{64}$$

- Let an 8-character long password. Assume that it uses only small-case letters i.e. a–z. How many are the possible passwords?

$$\text{number of passwords } = 26^8 \approx 2^{36}$$

Passwords

Comparing cryptographic keys to passwords

- Let a randomly chosen 64-bit key. How many are the possible keys?

$$\text{number of keys } = 2^{64}$$

- Let an 8-character long password. Assume that it uses only small-case letters i.e. a–z. How many are the possible passwords?

$$\text{number of passwords } = 26^8 \approx 2^{36}$$

- Assume that we use some part of Unicode e.g. we have 256 distinct characters to choose from. How many are the possible passwords?

$$\text{number of passwords } = 256^8 = 2^{64}$$

Passwords

Comparing cryptographic keys to passwords

- Let a randomly chosen 64-bit key. How many are the possible keys?

$$\text{number of keys } = 2^{64}$$

- Let an 8-character long password. Assume that it uses only small-case letters i.e. a–z. How many are the possible passwords?

$$\text{number of passwords } = 26^8 \approx 2^{36}$$

- Assume that we use some part of Unicode e.g. we have 256 distinct characters to choose from. How many are the possible passwords?

$$\text{number of passwords } = 256^8 = 2^{64}$$

- It appears that brute-forcing a key and guessing a password has the same complexity

Passwords

- However users must remember their password thus they tend to use easy phrases for them:

  `password, 12345678, 88888888, 7-3-1988, Amsterdam, ...`

Passwords

- However users must remember their password thus they tend to use easy phrases for them:

  `password, 12345678, 88888888, 7-3-1988, Amsterdam, ...`

- Users will rarely choose uniformly random passwords such as:

  `5gT9mAw@, hTp05M!], ...`

Passwords

- However users must remember their password thus they tend to use easy phrases for them:

  `password, 12345678, 88888888, 7-3-1988, Amsterdam, ...`

- Users will rarely choose uniformly random passwords such as:

  `5gT9mAw@, hTp05M!], ...`

- A smart attacker will not try to guess all possible passwords. She will use a **dictionary** of e.g. $2^{20} \approx 10^6$ common words

Passwords

- However users must remember their password thus they tend to use easy phrases for them:

  `password, 12345678, 88888888, 7-3-1988, Amsterdam, ...`

- Users will rarely choose uniformly random passwords such as:

  `5gT9mAw@, hTp05M!], ...`

- A smart attacker will not try to guess all possible passwords. She will use a **dictionary** of e.g. $2^{20} \approx 10^6$ common words
- The probability of guessing a random 64-bit key with 2^{20} attempts is very low since $2^{20}/2^{64} = 2^{-44}$

Passwords

- However users must remember their password thus they tend to use easy phrases for them:

  `password, 12345678, 88888888, 7-3-1988, Amsterdam, ...`

- Users will rarely choose uniformly random passwords such as:

  `5gT9mAw@, hTp05M!], ...`

- A smart attacker will not try to guess all possible passwords. She will use a **dictionary** of e.g. $2^{20} \approx 10^6$ common words
- The probability of guessing a random 64-bit key with 2^{20} attempts is very low since $2^{20}/2^{64} = 2^{-44}$
- However, the probability of guessing the password with a 2^{20}-word dictionary can be fairly high due to the non-randomness

Passwords

- Experiments showed that about 30% of human-generated passwords can be easy to guess. Generating passwords using a passphrase showed improvements, making the easy-to-guess passwords 10%.

  e.g. `These days I feel like studying Linear Algebra`

Passwords

- Experiments showed that about 30% of human-generated passwords can be easy to guess. Generating passwords using a passphrase showed improvements, making the easy-to-guess passwords 10%.

  e.g. `These days I feel like studying Linear Algebra`

- Automated tools have been developed that can recover passwords with dictionary and brute-force attacks
- John the Ripper, L0phtCrack
  `https://www.openwall.com/john/`
  `https://l0phtcrack.gitlab.io/`

Passwords

- Experiments showed that about 30% of human-generated passwords can be easy to guess. Generating passwords using a passphrase showed improvements, making the easy-to-guess passwords 10%.

  e.g. `These days I feel like studying Linear Algebra`

- Automated tools have been developed that can recover passwords with dictionary and brute-force attacks
- John the Ripper, L0phtCrack
  `https://www.openwall.com/john/`
  `https://l0phtcrack.gitlab.io/`

- User compliance to a password policy is hard to achieve
  - Select randomly a password for the user, instead of letting them choose
  - Require special characters, capital letters, impose minimal password length, avoid the username in the password etc.

Passwords

- Experiments showed that about 30% of human-generated passwords can be easy to guess. Generating passwords using a passphrase showed improvements, making the easy-to-guess passwords 10%.

  e.g. `These days I feel like studying Linear Algebra`

- Automated tools have been developed that can recover passwords with dictionary and brute-force attacks
- John the Ripper, L0phtCrack
  `https://www.openwall.com/john/`
  `https://l0phtcrack.gitlab.io/`

- User compliance to a password policy is hard to achieve
  - Select randomly a password for the user, instead of letting them choose
  - Require special characters, capital letters, impose minimal password length, avoid the username in the password etc.
- Users may still circumvent the password policy

  `Amsterdam1, Amsterdam2, Amsterdam3, Amsterdam1, ...`

Password Verification and Attacks

Password Verification and Attacks

- How does a system verify if a password is correct?
- We must compare the password against *something*

Password Verification and Attacks

- How does a system verify if a password is correct?
- We must compare the password against *something*

- **Naive comparison method.** Store all userids and passwords in a file then perform the comparison. If successful, grant access.
- What happens if you get hacked? You loose all passwords immediately!

Password Verification and Attacks

- How does a system verify if a password is correct?
- We must compare the password against *something*

- **Naive comparison method.** Store all userids and passwords in a file then perform the comparison. If successful, grant access.
- What happens if you get hacked? You loose all passwords immediately!

- **Hash-based comparison.** Once the user registers and chooses a password x the system will apply a hash function to it. Instead of storing the password x, the system will store its hash value h.

$$\text{Compute: } h = hash(x), \quad \text{Store in database: } (userid, h)$$

Password Verification and Attacks

- How does a system verify if a password is correct?
- We must compare the password against *something*

- **Naive comparison method.** Store all userids and passwords in a file then perform the comparison. If successful, grant access.
- What happens if you get hacked? You loose all passwords immediately!

- **Hash-based comparison.** Once the user registers and chooses a password x the system will apply a hash function to it. Instead of storing the password x, the system will store its hash value h.

$$\text{Compute: } h = hash(x), \quad \text{Store in database: } (userid, h)$$

- To verify a user password x' the system will hash it and compare it to the hashed value in the database

$$\text{Compute: } h' = hash(x'), \quad \text{Compare: } h' == h$$

Password Verification and Attacks

- Notice that every time you reset your password you do not get the old one back, you must create a new one. By hashing the passwords we avoid storing directly and thus leaking them.
- **Preimage resistance.** Given h it is computationally difficult to find password x such that $hash(x) = h$

Password Verification and Attacks

- Notice that every time you reset your password you do not get the old one back, you must create a new one. By hashing the passwords we avoid storing directly and thus leaking them.
- **Preimage resistance.** Given h it is computationally difficult to find password x such that $hash(x) = h$
- Still, assume an attacker that launches a dictionary attack and let a dictionary $\mathcal{D}$ that contains the b most common passwords. The attack cannot use $\mathcal{D}$ as is due to hashing.

$$\mathcal{D} = \{x_1, x_2, \ldots, x_b\}$$

Password Verification and Attacks

- Notice that every time you reset your password you do not get the old one back, you must create a new one. By hashing the passwords we avoid storing directly and thus leaking them.
- **Preimage resistance.** Given h it is computationally difficult to find password x such that $hash(x) = h$
- Still, assume an attacker that launches a dictionary attack and let a dictionary $\mathcal{D}$ that contains the b most common passwords. The attack cannot use $\mathcal{D}$ as is due to hashing.

$$\mathcal{D} = \{x_1, x_2, \ldots, x_b\}$$

- To avoid a slow **online hash computation** of the dictionary passwords, the attacker can hash them in advance, i.e. create a **lookup table**

$$\mathcal{H} = \{hash(x_1), hash(x_2), \ldots, hash(x_b)\}$$

Password Verification and Attacks

- Notice that every time you reset your password you do not get the old one back, you must create a new one. By hashing the passwords we avoid storing directly and thus leaking them.
- **Preimage resistance.** Given h it is computationally difficult to find password x such that $hash(x) = h$
- Still, assume an attacker that launches a dictionary attack and let a dictionary $\mathcal{D}$ that contains the b most common passwords. The attack cannot use $\mathcal{D}$ as is due to hashing.

$$\mathcal{D} = \{x_1, x_2, \ldots, x_b\}$$

- To avoid a slow **online hash computation** of the dictionary passwords, the attacker can hash them in advance, i.e. create a **lookup table**

$$\mathcal{H} = \{hash(x_1), hash(x_2), \ldots, hash(x_b)\}$$

- Now the dictionary attack amounts to a single table lookup in $\mathcal{H}$ thus it is much faster than computing hashes on-the-fly

Password Verification and Attacks

- Using high-speed implementations and parallel processing clusters, we can construct very large hashed lookup tables
- For any given hash function (e.g. MD5, SHA256, Keccak) these tables need to be constructed only **once**

Password Verification and Attacks

- Using high-speed implementations and parallel processing clusters, we can construct very large hashed lookup tables
- For any given hash function (e.g. MD5, SHA256, Keccak) these tables need to be constructed only **once**

- Thus when the hash tables are precomputed, the dictionary attack on the hash-based verification is trivial
- Can we prevent that?

Password Verification and Attacks

Salt values

- Instead of hashing the password directly we will first generate a **salt value**, a non-secret value for each password

Password Verification and Attacks

Salt values

- Instead of hashing the password directly we will first generate a **salt value**, a non-secret value for each password
- We will hash the password x concatenated with its salt

$$\text{Compute: } h = hash(x||salt), \quad \text{Store in database: } (userid, h, salt)$$

- Note that the salt is stored in unencrypted form since it is not secret and that every user has a different salt

Password Verification and Attacks

Salt values

- Instead of hashing the password directly we will first generate a **salt value**, a non-secret value for each password
- We will hash the password x concatenated with its salt

$$\text{Compute: } h = hash(x||salt), \quad \text{Store in database: } (userid, h, salt)$$

- Note that the salt is stored in unencrypted form since it is not secret and that every user has a different salt
- To verify a user password x' the system will append the salt, hash and compare

$$\text{Compute: } h' = hash(x'||salt), \quad \text{Compare: } h == h'$$

Password Verification and Attacks

Salt values

- Instead of hashing the password directly we will first generate a **salt value**, a non-secret value for each password
- We will hash the password x concatenated with its salt

$$\text{Compute: } h = hash(x||salt), \quad \text{Store in database: } (userid, h, salt)$$

- Note that the salt is stored in unencrypted form since it is not secret and that every user has a different salt
- To verify a user password x' the system will append the salt, hash and compare

$$\text{Compute: } h' = hash(x'||salt), \quad \text{Compare: } h == h'$$

- Note that now we cannot precompute all the hashes in a password dictionary, since we have to first concatenate every password with a random salt

Password Verification and Attacks

Salt values

- Instead of hashing the password directly we will first generate a **salt value**, a non-secret value for each password
- We will hash the password x concatenated with its salt

$$\text{Compute: } h = hash(x||salt), \quad \text{Store in database: } (userid, h, salt)$$

- Note that the salt is stored in unencrypted form since it is not secret and that every user has a different salt
- To verify a user password x' the system will append the salt, hash and compare

$$\text{Compute: } h' = hash(x'||salt), \quad \text{Compare: } h == h'$$

- Note that now we cannot precompute all the hashes in a password dictionary, since we have to first concatenate every password with a random salt
- The attacker must precompute the table for all salts in the database (number of different salts $\approx$ number of users) or perform online hash computations

Password Verification and Attacks

Exercise: Estimating the attack effort

- Let 8-character passwords using 128 choices per character i.e. there exist $128^8 = 2^{56}$ possible passwords
- The user database contains 2^{10} hashed passwords
- The attacker has a dictionary with 2^{20} common hashed values
- The attacker expects 25% of the passwords to be in the dictionary

  We want to estimate the average effort to to crack certain passwords in various cases (Case 1 until 3). We measure the effort in "number of hashes to compute" i.e. table lookups and comparisons are free.

Password Verification and Attacks

Case 1. What is the effort to recover a specific user password x from its hash h *without using a dictionary of likely passwords*.

Password Verification and Attacks

Case 1. What is the effort to recover a specific user password x from its hash h *without using a dictionary of likely passwords*.

- This is equivalent to an exhaustive search across all possible 2^{56} passwords

$$\text{worst case effort} = \text{number of all possible hashes} =$$

$$2^{56} \text{ hashes}$$

$$\text{average effort} = \text{number of hashes computed s.t. probability} \geq 0.5 =$$

$$\frac{2^{56}}{2} = 2^{55} \text{ hashes}$$

- Since we do not use the dictionary, using salt (or not using salt) does not change the situation

Password Verification and Attacks

Case 2. What is the average effort to recover a specific user password x from its hash h when *using a dictionary* of 2^{20} likely passwords.

Password Verification and Attacks

Case 2. What is the average effort to recover a specific user password x from its hash h when *using a dictionary* of 2^{20} likely passwords.

- **Case 2A.** Let's assume 'unsalted' passwords i.e. the attacker can use the precomputed dictionary of hashed values

Password Verification and Attacks

Case 2. What is the average effort to recover a specific user password x from its hash h when *using a dictionary* of 2^{20} likely passwords.

- **Case 2A.** Let's assume 'unsalted' passwords i.e. the attacker can use the precomputed dictionary of hashed values
- With probability 25% the password will appear in the dictionary and can be recovered with a table lookup. With probability 75% it will not, requiring to compute the hashes.

$$\text{worst-case effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{56} \approx 2^{55.6} \text{ hashes}$$

$$\text{average effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{55} \approx 2^{54.6} \text{ hashes}$$

This excludes the one-time dictionary building cost.

Password Verification and Attacks

Case 2. What is the average effort to recover a specific user password x from its hash h when *using a dictionary* of 2^{20} likely passwords.

- **Case 2A.** Let's assume 'unsalted' passwords i.e. the attacker can use the precomputed dictionary of hashed values
- With probability 25% the password will appear in the dictionary and can be recovered with a table lookup. With probability 75% it will not, requiring to compute the hashes.

$$\text{worst-case effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{56} \approx 2^{55.6} \text{ hashes}$$

$$\text{average effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{55} \approx 2^{54.6} \text{ hashes}$$

  This excludes the one-time dictionary building cost.

- **Case 2B.** Let's assume 'salted' passwords i.e. the attacker must compute from scratch the dictionary for the given *salt* of the password x

$$\text{worst-case effort} = \frac{1}{4} * 2^{20} + \frac{3}{4} * 2^{56} \approx 2^{55.6} \text{ hashes}$$

Password Verification and Attacks

Case 2. What is the average effort to recover a specific user password x from its hash h when *using a dictionary* of 2^{20} likely passwords.

- **Case 2A.** Let's assume 'unsalted' passwords i.e. the attacker can use the precomputed dictionary of hashed values
- With probability 25% the password will appear in the dictionary and can be recovered with a table lookup. With probability 75% it will not, requiring to compute the hashes.

$$\text{worst-case effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{56} \approx 2^{55.6} \text{ hashes}$$

$$\text{average effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{55} \approx 2^{54.6} \text{ hashes}$$

This excludes the one-time dictionary building cost.

- **Case 2B.** Let's assume 'salted' passwords i.e. the attacker must compute from scratch the dictionary for the given *salt* of the password x

$$\text{worst-case effort} = \frac{1}{4} * 2^{20} + \frac{3}{4} * 2^{56} \approx 2^{55.6} \text{ hashes}$$

$$\text{average effort} = \frac{1}{4} * 2^{19} + \frac{3}{4} * 2^{55} \approx 2^{54.6} \text{ hashes}$$

Password Verification and Attacks

Case 2. What is the average effort to recover a specific user password x from its hash h when *using a dictionary* of 2^{20} likely passwords.

- **Case 2A.** Let's assume 'unsalted' passwords i.e. the attacker can use the precomputed dictionary of hashed values
- With probability 25% the password will appear in the dictionary and can be recovered with a table lookup. With probability 75% it will not, requiring to compute the hashes.

$$\text{worst-case effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{56} \approx 2^{55.6} \text{ hashes}$$

$$\text{average effort} = \frac{1}{4} * 0 + \frac{3}{4} * 2^{55} \approx 2^{54.6} \text{ hashes}$$

This excludes the one-time dictionary building cost.

- **Case 2B.** Let's assume 'salted' passwords i.e. the attacker must compute from scratch the dictionary for the given *salt* of the password x

$$\text{worst-case effort} = \frac{1}{4} * 2^{20} + \frac{3}{4} * 2^{56} \approx 2^{55.6} \text{ hashes}$$

$$\text{average effort} = \frac{1}{4} * 2^{19} + \frac{3}{4} * 2^{55} \approx 2^{54.6} \text{ hashes}$$

- The effort is close to the effort for Case 1 (no dictionary), however the attacker can stop after trying all values in the dictionary and still have a 25% probability of success.

Password Verification and Attacks

Case 3. So far we focused on a single password, yet the database contains 2^{10} passwords. What is the average effort to recover *any* password from the 2^{10} passwords in the user database, without using the dictionary.

Password Verification and Attacks

Case 3. So far we focused on a single password, yet the database contains 2^{10} passwords. What is the average effort to recover *any* password from the 2^{10} passwords in the user database, without using the dictionary.

- **Case 3A.** Let's assume 'unsalted' passwords
- Let 2^{10} distinct password hashes $v_1, v_2, \ldots, v_{2^{10}}$ stored in the database

Password Verification and Attacks

Case 3. So far we focused on a single password, yet the database contains 2^{10} passwords. What is the average effort to recover *any* password from the 2^{10} passwords in the user database, without using the dictionary.

- **Case 3A.** Let's assume 'unsalted' passwords
- Let 2^{10} distinct password hashes $v_1, v_2, \ldots, v_{2^{10}}$ stored in the database
- The attacker can perform an exhaustive search over the passwords x_i, $i = 1, 2, \ldots, 2^{56}$. Every $hash(x_i)$ will be compared to every hash v_j in the database.

$$\text{average effort} = \frac{2^{55}}{2^{10}} = 2^{45} \text{ hashes} + \text{comparison cost}$$

- The cost of computing a hash function typically dominates the cost of comparing to all database hashes, i.e. comparison cost ≈ 0

Password Verification and Attacks

Case 3. So far we focused on a single password, yet the database contains 2^{10} passwords. What is the average effort to recover *any* password from the 2^{10} passwords in the user database, without using the dictionary.

- **Case 3A.** Let's assume 'unsalted' passwords
- Let 2^{10} distinct password hashes $v_1, v_2, \ldots, v_{2^{10}}$ stored in the database
- The attacker can perform an exhaustive search over the passwords x_i, $i = 1, 2, \ldots, 2^{56}$. Every $hash(x_i)$ will be compared to every hash v_j in the database.

$$\text{average effort} = \frac{2^{55}}{2^{10}} = 2^{45} \text{ hashes} + \text{comparison cost}$$

- The cost of computing a hash function typically dominates the cost of comparing to all database hashes, i.e. comparison cost ≈ 0

- **Case 3B.** Let's assume 'salted' passwords
- Hash every password x_i together with every possible salt $salt_j$ in the database

$$\text{for all } i = 1, 2, \ldots, 2^{56} \text{ and all } j = 1, 2, \ldots, 2^{10}$$

$$\text{compute } hash(x_i || salt_j) \text{ and compare to all } v_j \text{ in database}$$

$$\text{average effort} = 2^{10} \frac{2^{55}}{2^{10}} = 2^{55} \text{ hashes}$$

Entropy

Entropy

- To quantify the uncertainty of a password we will use the concept of **entropy**

Entropy

- To quantify the uncertainty of a password we will use the concept of **entropy**
- Let a **discrete random variable** X that describes a probabilistic experiment

  e.g. random variable X describes the roll of a 6-sided die
- The random variable X ranges over the set $\mathcal{X}$

  e.g. $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$

Entropy

- To quantify the uncertainty of a password we will use the concept of **entropy**
- Let a **discrete random variable** X that describes a probabilistic experiment

  e.g. random variable X describes the roll of a 6-sided die
- The random variable X ranges over the set $\mathcal{X}$

  e.g. $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$
- Attached to X is a **probability mass function** (p.m.f.)
  $p(x) = P(X = x)$ for all $x \in \mathcal{X}$

  e.g. in the case of a fair die: $p(1) = p(2) = \cdots = p(6) = \dfrac{1}{6}$

Entropy

- To quantify the uncertainty of a password we will use the concept of **entropy**
- Let a **discrete random variable** X that describes a probabilistic experiment

  e.g. random variable X describes the roll of a 6-sided die
- The random variable X ranges over the set $\mathcal{X}$

  e.g. $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$
- Attached to X is a **probability mass function** (p.m.f.) $p(x) = P(X = x)$ for all $x \in \mathcal{X}$

  e.g. in the case of a fair die: $p(1) = p(2) = \cdots = p(6) = \frac{1}{6}$

  e.g. in the case of a cheating die: $p(1) = p(2) = p(3) = p(4) = \frac{1}{6}$

  but $p(5) = \frac{1}{12}$ and $p(6) = \frac{1}{4}$ making a 6-roll more likely

Entropy

Entropy. The entropy $H(X)$ of a discrete random variable X is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

Entropy

Entropy. The entropy $H(X)$ of a discrete random variable X is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

- Entropy is expressed in **bits**
- A higher number of bits implies more uncertainty

Entropy

Entropy. The entropy $H(X)$ of a discrete random variable X is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

- Entropy is expressed in **bits**
- A higher number of bits implies more uncertainty

- When computing the entropy $H(X)$ we do not consider cases of x with zero probability because of $0 * log_2 0$ values
- It holds that $H(X) \geq 0$ always

Entropy

Example 1: Fair die roll

.

- Compute the entropy $H(X)$ where X is the fair die roll example

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) = -\sum_{x \in \{1,2,3,4,5,6\}} p(x) \log_2 p(x) =$$

$$-\sum_{x \in \{1,2,3,4,5,6\}} \frac{1}{6} \log_2 \frac{1}{6} = -\left(\frac{1}{6} \log_2 \frac{1}{6} + \cdots + \frac{1}{6} \log_2 \frac{1}{6} \right) =$$

$$-\log_2 \frac{1}{6} = 2.585 \text{ bits}$$

Entropy

Example 2: Cheating die roll

- Compute the entropy $H(X)$ where X is the cheating die roll example

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) = -\sum_{x \in \{1,2,3,4,5,6\}} p(x) \log_2 p(x) =$$

$$-\left(p(1) \log_2 p(1) + \cdots + p(4) \log_2 p(4) + p(5) \log_2 p(5) + p(6) \log_2 p(6)\right) =$$

$$-\left(\frac{4}{6} \log_2 \frac{1}{6} + \frac{1}{12} \log_2 \frac{1}{12} + \frac{1}{4} \log_2 \frac{1}{4}\right) = 2.5221 \text{ bits}$$

Entropy

Example 2: Cheating die roll

- Compute the entropy $H(X)$ where X is the cheating die roll example

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) = -\sum_{x \in \{1,2,3,4,5,6\}} p(x) \log_2 p(x) =$$

$$-\left(p(1) \log_2 p(1) + \cdots + p(4) \log_2 p(4) + p(5) \log_2 p(5) + p(6) \log_2 p(6)\right) =$$

$$-\left(\frac{4}{6} \log_2 \frac{1}{6} + \frac{1}{12} \log_2 \frac{1}{12} + \frac{1}{4} \log_2 \frac{1}{4}\right) = 2.5221 \text{ bits}$$

- The entropy of the 'cheating die' is less than the entropy of the 'fair die', i.e. the fair die has more uncertainty
- In general, entropy is maximized when all values of X are equiprobable

$$\text{Maximum entropy} \quad H(X) = \log_2 p, \quad \text{where } p \text{ the probability of all } x \in \mathcal{X}$$

Entropy

Example 2: Cheating die roll

- Compute the entropy $H(X)$ where X is the cheating die roll example

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\log_2 p(x) = -\sum_{x\in\{1,2,3,4,5,6\}} p(x)\log_2 p(x) =$$

$$-\left(p(1)\log_2 p(1) + \cdots + p(4)\log_2 p(4) + p(5)\log_2 p(5) + p(6)\log_2 p(6)\right) =$$

$$-\left(\frac{4}{6}\log_2\frac{1}{6} + \frac{1}{12}\log_2\frac{1}{12} + \frac{1}{4}\log_2\frac{1}{4}\right) = 2.5221 \text{ bits}$$

- The entropy of the 'cheating die' is less than the entropy of the 'fair die', i.e. the fair die has more uncertainty
- In general, entropy is maximized when all values of X are equiprobable

$$\text{Maximum entropy} \quad H(X) = \log_2 p, \quad \text{where } p \text{ the probability of all } x \in \mathcal{X}$$

- We will now use entropy to estimate the strength of passwords

Entropy

Example 3: Uniformly chosen passwords

- Let a 4-character password
- Assume a perfect user that chooses every password character according to a uniformly random distribution over the set $\mathcal{X} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$
- Compute the entropy of a single character of the password

Entropy

Example 3: Uniformly chosen passwords

- Let a 4-character password
- Assume a perfect user that chooses every password character according to a uniformly random distribution over the set $\mathcal{X} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$
- Compute the entropy of a single character of the password

- Let random variable X that represents the choice of a password character
- The r.v. X is uniformly distributed thus:

$$P(X = x) = \frac{1}{|\mathcal{X}|}$$

$$\text{i.e.} \quad P(X = \mathtt{a}) = P(X = \mathtt{b}) = \cdots = P(X = \mathtt{z}) = \frac{1}{26}$$

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) = -\sum_{x \in \{\mathtt{a}, \ldots, \mathtt{z}\}} \frac{1}{26} \log_2 \frac{1}{26} = 4.70 \text{ bits}$$

Entropy

Example 3: Uniformly chosen passwords

- Compute the entropy of the 4-character password

Entropy

Example 3: Uniformly chosen passwords

- Compute the entropy of the 4-character password
- Let random vector **Y** that represents the 4 characters

$$Y = [X_1 \;\; X_2 \;\; X_3 \;\; X_4]$$

- The characters X_1, X_2, X_3, X_4 are chosen independently thus:

$$P(Y = y) = p(y) = P([X_1 \;\; X_2 \;\; X_3 \;\; X_4] = [x_1 \;\; x_2 \;\; x_3 \;\; x_4]) =$$

$$P(x_1)P(x_2)P(x_3)P(x_4) = \left(\frac{1}{|\mathcal{X}|}\right)^4 = \left(\frac{1}{26}\right)^4$$

Entropy

Example 3: Uniformly chosen passwords

- Compute the entropy of the 4-character password
- Let random vector **Y** that represents the 4 characters

$$Y = [X_1 \;\; X_2 \;\; X_3 \;\; X_4]$$

- The characters X_1, X_2, X_3, X_4 are chosen independently thus:

$$P(Y = y) = p(y) = P([X_1 \;\; X_2 \;\; X_3 \;\; X_4] = [x_1 \;\; x_2 \;\; x_3 \;\; x_4]) =$$

$$P(x_1)P(x_2)P(x_3)P(x_4) = \left(\frac{1}{|\mathcal{X}|}\right)^4 = \left(\frac{1}{26}\right)^4$$

- The entropy is computed as a sum over 26^4 equiprobable passwords **y**

$$H(Y) = -\sum_{y \in \mathcal{Y}} p(\mathbf{y}) \log_2 p(y) = -\log_2 \left(\frac{1}{26}\right)^4 = 18.80 \text{ bits}$$

Entropy

Example 4: Biased passwords

- Assume a non-ideal user that does not choose the password characters randomly
- We know that this user is much more likely to pick a 4-character password that is an actual word
- There are approximately 4000 common 4-character words in the English language and the user has a 70% probability of choosing one of them randomly
- Compute the entropy of the 4-character password

Entropy

Example 4: Biased passwords

- Assume a non-ideal user that does not choose the password characters randomly
- We know that this user is much more likely to pick a 4-character password that is an actual word
- There are approximately 4000 common 4-character words in the English language and the user has a 70% probability of choosing one of them randomly
- Compute the entropy of the 4-character password

- Let the set $\mathcal{Y}$ of all possible passwords and we split it to a set $\mathcal{C}$ of 4000 common passwords and a set $\mathcal{U}$ of uncommon passwords (the rest)

$$\mathcal{Y} = \mathcal{C} \cup \mathcal{U}, \quad |\mathcal{C}| = 4000, \ \ |\mathcal{U}| = 26^4 - 4000$$

Entropy

Example 4: Biased passwords

- Assume a non-ideal user that does not choose the password characters randomly
- We know that this user is much more likely to pick a 4-character password that is an actual word
- There are approximately 4000 common 4-character words in the English language and the user has a 70% probability of choosing one of them randomly
- Compute the entropy of the 4-character password

- Let the set $\mathcal{Y}$ of all possible passwords and we split it to a set $\mathcal{C}$ of 4000 common passwords and a set $\mathcal{U}$ of uncommon passwords (the rest)

$$\mathcal{Y} = \mathcal{C} \cup \mathcal{U}, \quad |\mathcal{C}| = 4000, \ |\mathcal{U}| = 26^4 - 4000$$

- Passwords y in common set $\mathcal{C}$ are more likely than set $\mathcal{U}$

$$p(y \in \mathcal{C}) = 0.7, \quad p(y \in \mathcal{U}) = 0.3$$

Entropy

Let $\mathcal{C} = \{y_1, y_2, \ldots, y_{4000}\}$ passwords and assume that they are all equally likely

$$p(y \in \mathcal{C}) = 0.7 \iff p(y_1 \cup y_2 \cup \cdots \cup y_{4000}) = 0.7 \iff$$

$$p(y_1) + p(y_2) + \cdots + p(y_{4000}) = 0.7 \iff 4000p = 0.7 \iff p = 0.7/4000$$

Entropy

Let $\mathcal{C} = \{y_1, y_2, \ldots, y_{4000}\}$ passwords and assume that they are all equally likely

$$p(y \in \mathcal{C}) = 0.7 \iff p(y_1 \cup y_2 \cup \cdots \cup y_{4000}) = 0.7 \iff$$

$$p(y_1) + p(y_2) + \cdots + p(y_{4000}) = 0.7 \iff 4000p = 0.7 \iff p = 0.7/4000$$

Thus we have: $$p(y) = \begin{cases} 0.7/4000, & \text{if } y \in \mathcal{C} \\ 0.3/(26^4 - 4000), & \text{if } y \in \mathcal{U} \end{cases}$$

Entropy

Let $\mathcal{C} = \{y_1, y_2, \ldots, y_{4000}\}$ passwords and assume that they are all equally likely

$$p(y \in \mathcal{C}) = 0.7 \iff p(y_1 \cup y_2 \cup \cdots \cup y_{4000}) = 0.7 \iff$$

$$p(y_1) + p(y_2) + \cdots + p(y_{4000}) = 0.7 \iff 4000p = 0.7 \iff p = 0.7/4000$$

$$\text{Thus we have:} \quad p(y) = \begin{cases} 0.7/4000, & \text{if } y \in \mathcal{C} \\ 0.3/(26^4 - 4000), & \text{if } y \in \mathcal{U} \end{cases}$$

$$H(Y) = -\sum_{y \in \mathcal{Y}} p(y) \log_2 p(y) = -\left(\sum_{y \in \mathcal{C}} p(y) \log_2 p(y) + \sum_{y \in \mathcal{U}} p(y) \log_2 p(y) \right) =$$

$$-\left(\sum_{i=1}^{4000} \frac{0.7}{4000} \log_2 \frac{0.7}{4000} + \sum_{i=1}^{26^4 - 4000} \frac{0.3}{26^4 - 4000} \log_2 \frac{0.3}{26^4 - 4000} \right) =$$

14.89 bits $<$ 18.80 bits for the uniform password example

Entropy

- Entropy is a standard metric but it assumes that the attacker tries passwords in a random way until the correct one is found
- Attackers are smarter than this! First they will try the most common passwords and if they fail, they will try less common choices

Entropy

- Entropy is a standard metric but it assumes that the attacker tries passwords in a random way until the correct one is found
- Attackers are smarter than this! First they will try the most common passwords and if they fail, they will try less common choices

Cumulative probability of success (CPS). Assume an attacker that has identified a password guessing strategy i.e. to guess among n passwords, he produced an ordering of passwords from most likely to least likely.

$$x_1, x_2, x_3, \ldots, x_n, \quad \text{such that} \quad p(x_1) \geq p(x_2) \geq p(x_3) \geq \cdots \geq p(x_n)$$

Entropy

- Entropy is a standard metric but it assumes that the attacker tries passwords in a random way until the correct one is found
- Attackers are smarter than this! First they will try the most common passwords and if they fail, they will try less common choices

Cumulative probability of success (CPS). Assume an attacker that has identified a password guessing strategy i.e. to guess among n passwords, he produced an ordering of passwords from most likely to least likely.

$$x_1, x_2, x_3, \ldots, x_n, \quad \text{such that} \quad p(x_1) \geq p(x_2) \geq p(x_3) \geq \cdots \geq p(x_n)$$

We define $CPS(b)$ as follows:

$$CPS(b) = \sum_{i=1}^{b} p(x_i)$$

Entropy

- Entropy is a standard metric but it assumes that the attacker tries passwords in a random way until the correct one is found
- Attackers are smarter than this! First they will try the most common passwords and if they fail, they will try less common choices

Cumulative probability of success (CPS). Assume an attacker that has identified a password guessing strategy i.e. to guess among n passwords, he produced an ordering of passwords from most likely to least likely.

$$x_1, x_2, x_3, \ldots, x_n, \quad \text{such that} \quad p(x_1) \geq p(x_2) \geq p(x_3) \geq \cdots \geq p(x_n)$$

We define $CPS(b)$ as follows:

$$CPS(b) = \sum_{i=1}^{b} p(x_i)$$

CPS(b) gives the probability of success of our strategy when trying b out of n passwords. Note that $CPS(n) = 1$ i.e. trying all passwords guarantees success.

Entropy

Example 5: CPS computation on a password guessing strategy

- Assume that a non-ideal user chooses a 4-character password i.e. $n = 26^4$
- Prior experience shows that users choose passwords from the following 3 disjoint sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$
- The user may choose (with probability 50%) password from set $\mathcal{A}$ that contains only 0.1% of the passwords
- The user may choose (with probability 30%) password from set $\mathcal{B}$ that contains only 2% of the passwords
- The user may choose (with probability 20%) choose a password from the set $\mathcal{C}$ that contains the remaining 97.9% of the passwords

Entropy

Example 5: CPS computation on a password guessing strategy

- Sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$ contain 456, 9139 and 447381 passwords respectively
- The guessing strategy will start guessing passwords from $\mathcal{A}$. If it fails it continues with set $\mathcal{B}$ and if it fails again it tries set $\mathcal{C}$.
- Individual passwords in sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$ have probability of 0.5/456, 0.3/9139 and 0.2/447381 respectively

Entropy

Example 5: CPS computation on a password guessing strategy

- Sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$ contain 456, 9139 and 447381 passwords respectively
- The guessing strategy will start guessing passwords from $\mathcal{A}$. If it fails it continues with set $\mathcal{B}$ and if it fails again it tries set $\mathcal{C}$.
- Individual passwords in sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$ have probability of 0.5/456, 0.3/9139 and 0.2/447381 respectively

- Compute $CPS(100)$, $CPS(10^3)$ and $CPS(10^4)$

$$CPS(100) = \sum_{i=1}^{100} p_i = \sum_{i=1}^{100} \frac{0.5}{456} \approx 0.10$$

$$CPS(10^3) = \sum_{i=1}^{10^3} p_i = \sum_{i=1}^{456} \frac{0.5}{456} + \sum_{i=457}^{10^3} \frac{0.3}{9139} \approx 0.51$$

$$CPS(10^4) = \sum_{i=1}^{10^4} p_i = \sum_{i=1}^{456} \frac{0.5}{456} + \sum_{i=457}^{9595} \frac{0.3}{9139} + \sum_{i=9596}^{10^4} \frac{0.2}{447381} \approx 0.80$$

Rainbow Tables

Rainbow Tables

- Let an 8-character password, whose digits are chosen over 256 values

$$\text{number of passwords } = 256^8 = 2^{64}$$

- We managed to gain access to the `/etc/shadow` file and obtained the hashed value h of this password
- We will start a brute-force attack to recover the password from its hash h

$$\text{We have the hashed value:} \quad h = hash(x)$$

$$\text{We want the preimage:} \quad x = hash^{-1}(h)$$

Rainbow Tables

Attack extremas

1. **Online computation.** Iterate over 2^{64} passwords and hash each one until we find the password that hashes to the recovered value h
   - Modern Intel/AMD processor have instruction set extensions for the $SHA256$ hash function
     `https://bench.cr.yp.to/`
   - Such implementations can reach 1.6 clock cycles per byte
   - Hashing 8 bytes at 4GHz takes:
     $8 * 1.6/(4 * 10^9) = 3.2 * 10^{-9}$ seconds
   - Total time is approximately $1.6 * 10^6$ hours i.e. 182 years

Rainbow Tables

Attack extremas

1. **Online computation.** Iterate over 2^{64} passwords and hash each one until we find the password that hashes to the recovered value h
   - Modern Intel/AMD processor have instruction set extensions for the $SHA256$ hash function
     `https://bench.cr.yp.to/`
   - Such implementations can reach 1.6 clock cycles per byte
   - Hashing 8 bytes at 4GHz takes:
     $8 * 1.6/(4 * 10^9) = 3.2 * 10^{-9}$ seconds
   - Total time is approximately $1.6 * 10^6$ hours i.e. 182 years

- It requires no memory except for the current hash value
- It needs a huge amount of time, so we should consider parallel processing

Rainbow Tables

2. **Lookup table (Offline computation).** To save time we can precompute the hash of 2^{64} passwords and store the hashed values in memory
    - This step can be done before any attack
    - Every value hashed with $SHA256$ is 32 bytes, thus storing them takes $32 * 2^{64} = 2^{69}$ bytes i.e. 590 exabytes

Rainbow Tables

2. **Lookup table (Offline computation).** To save time we can precompute the hash of 2^{64} passwords and store the hashed values in memory
    - This step can be done before any attack
    - Every value hashed with $SHA256$ is 32 bytes, thus storing them takes $32 * 2^{64} = 2^{69}$ bytes i.e. 590 exabytes

- It requires no computation at all, since the hash function table is precomputed
- It needs a huge amount of memory

Rainbow Tables

2. **Lookup table (Offline computation).** To save time we can precompute the hash of 2^{64} passwords and store the hashed values in memory

    - This step can be done before any attack
    - Every value hashed with $SHA256$ is 32 bytes, thus storing them takes $32 * 2^{64} = 2^{69}$ bytes i.e. 590 exabytes

- It requires no computation at all, since the hash function table is precomputed
- It needs a huge amount of memory

- **Rainbow tables** are a **time-memory tradeoff** between online and offline attacks
- The online computation can be viewed as an attack that needs $T = 2^{64}$ hash operations and $M = 1$ memory unit of 32 bytes
- The lookup table can be viewed as an attack than needs $T = 1$ lookup operation and $M = 2^{64}$ memory units of 32 bytes

Rainbow Tables

Rainbow table functions. To generate the rainbow tables we use two functions.

1. The hash function $hash(\cdot)$
   - We use the common hash function $SHA256(\cdot)$ and we choose password:

     $$x = 12345678$$

   - Converting the ASCII characters to hex we get:

     $$x = 31\ \ 32\ \ 33\ \ 34\ \ 35\ \ 36\ \ 37\ \ 38$$

   - Applying $SHA256$ on the password we get:

     $$h = \texttt{ef797c8118f02dfb649607dd5d3f8c76...}$$

     $$\texttt{...23048c9c063d532cc95c5ed7a898a64f}$$

Rainbow Tables

2. The reduction function $reduce(\cdot)$

   - Note that the output of $SHA256$ is 32 bytes

   $$h = \texttt{ef797c8118f02dfb649607dd5d3f8c76...}$$

   $$\texttt{...23048c9c063d532cc95c5ed7a898a64f}$$

   - We want to reduce this output to a new password x' that is 8 bytes long
   - To do this we can simply drop the last 24 bytes and generate a new password

   $$x' = reduce(h) = \texttt{ef 79 7c 81 18 f0 2d fb}$$

Rainbow Tables

- Putting together the $hash(\cdot)$ and $reduce(\cdot)$ functions we can define the function $f(\cdot)$ on password x

$$f(x) = reduce(hash(x))$$

Rainbow Tables

- Putting together the $hash(\cdot)$ and $reduce(\cdot)$ functions we can define the function $f(\cdot)$ on password x

$$f(x) = reduce(hash(x))$$

- Starting with password x_1, we can apply repeatedly the function $f(\cdot)$ to create a hash chain of length t

$$x_1 \xrightarrow{f} x_2 \xrightarrow{f} x_3 \xrightarrow{f} \ldots \xrightarrow{f} x_t$$

$$x_1 \xrightarrow{hash} h_1 \xrightarrow{reduce} x_2 \xrightarrow{hash} h_2 \xrightarrow{reduce} x_3 \ldots \xrightarrow{hash} h_{t-1} \xrightarrow{reduce} x_t$$

- We refer to x_1 as the **start point** and to x_t as the **end point**

Rainbow Tables

Precomputation Step (offline)

- We choose m different passwords as start points

$$\text{start points} \;= \{x_{1,1}, x_{2,1}, x_{3,1}, \ldots, x_{m,1}\}$$

- For every start point we create a hash chain of length t

$$x_{1,1} \xrightarrow{f} x_{1,2} \xrightarrow{f} x_{1,3} \xrightarrow{f} \ldots \xrightarrow{f} x_{1,t}$$

$$x_{2,1} \xrightarrow{f} x_{2,2} \xrightarrow{f} x_{2,3} \xrightarrow{f} \ldots \xrightarrow{f} x_{2,t}$$

$$x_{3,1} \xrightarrow{f} x_{3,2} \xrightarrow{f} x_{3,3} \xrightarrow{f} \ldots \xrightarrow{f} x_{3,t}$$

$$\ldots$$

$$x_{m,1} \xrightarrow{f} x_{m,2} \xrightarrow{f} x_{m,3} \xrightarrow{f} \ldots \xrightarrow{f} x_{m,t}$$

Rainbow Tables

- From this $m \times t$ table we only store the start and end points
- Thus every hash chain (row) is computed on the fly, keeping only the first and the last password

$$\mathbf{x_{1,1}} \xrightarrow{f} x_{1,2} \xrightarrow{f} x_{1,3} \xrightarrow{f} \ldots \xrightarrow{f} \mathbf{x_{1,t}}$$

$$\mathbf{x_{2,1}} \xrightarrow{f} x_{2,2} \xrightarrow{f} x_{2,3} \xrightarrow{f} \ldots \xrightarrow{f} \mathbf{x_{2,t}}$$

$$\mathbf{x_{3,1}} \xrightarrow{f} x_{3,2} \xrightarrow{f} x_{3,3} \xrightarrow{f} \ldots \xrightarrow{f} \mathbf{x_{3,t}}$$

$$\ldots$$

$$\mathbf{x_{m,1}} \xrightarrow{f} x_{m,2} \xrightarrow{f} x_{m,3} \xrightarrow{f} \ldots \xrightarrow{f} \mathbf{x_{m,t}}$$

Rainbow Tables

- From this $m \times t$ table we only store the start and end points
- Thus every hash chain (row) is computed on the fly, keeping only the first and the last password

$$
\begin{array}{c}
\mathbf{x_{1,1}} \xrightarrow{f} x_{1,2} \xrightarrow{f} x_{1,3} \xrightarrow{f} \dots \xrightarrow{f} \mathbf{x_{1,t}} \\
\mathbf{x_{2,1}} \xrightarrow{f} x_{2,2} \xrightarrow{f} x_{2,3} \xrightarrow{f} \dots \xrightarrow{f} \mathbf{x_{2,t}} \\
\mathbf{x_{3,1}} \xrightarrow{f} x_{3,2} \xrightarrow{f} x_{3,3} \xrightarrow{f} \dots \xrightarrow{f} \mathbf{x_{3,t}} \\
\dots \\
\mathbf{x_{m,1}} \xrightarrow{f} x_{m,2} \xrightarrow{f} x_{m,3} \xrightarrow{f} \dots \xrightarrow{f} \mathbf{x_{m,t}}
\end{array}
$$

- The precomputation step results in the tuple:

$$(\text{first password, last password}) = (x_{i,1}, x_{i,t}), \quad \text{for } i = 1, 2, 3, \dots, m$$

Rainbow Tables

Attack Step (online)

- We want to find which password results in hashed value h under $SHA256$
- We apply the $reduce(\cdot)$ function on h, resulting in password y

$$y = reduce(h)$$

Rainbow Tables

Attack Step (online)

- We want to find which password results in hashed value h under $SHA256$
- We apply the $reduce(\cdot)$ function on h, resulting in password y

$$y = reduce(h)$$

- **Case 1.** The password y is one of the end points of the rainbow table

$$y \in \{x_{1,t}, x_{2,t}, x_{3,t}, \ldots, x_{m,t}\}$$

Rainbow Tables

Attack Step (online)

- We want to find which password results in hashed value h under $SHA256$
- We apply the $reduce(\cdot)$ function on h, resulting in password y

$$y = reduce(h)$$

- **Case 1.** The password y is one of the end points of the rainbow table

$$y \in \{x_{1,t}, x_{2,t}, x_{3,t}, \ldots, x_{m,t}\}$$

- Assume that password y is found at the endpoint indexed j, i.e. $y = x_{j,t}$
- Looking at the chain we see that the hash value $h_{j,t}$ is reduced to password y

$$x_{j,1} \xrightarrow{hash} h_{j,1} \xrightarrow{reduce} x_{j,2} \xrightarrow{hash} \ldots \xrightarrow{reduce} x_{j,t-1} \xrightarrow{hash} \mathbf{h_{j,t-1}} \xrightarrow{reduce} \mathbf{x_{j,t}} = \mathbf{y}$$

Rainbow Tables

Attack Step (online)

- We want to find which password results in hashed value h under $SHA256$
- We apply the $reduce(\cdot)$ function on h, resulting in password y

$$y = reduce(h)$$

- **Case 1.** The password y is one of the end points of the rainbow table

$$y \in \{x_{1,t}, x_{2,t}, x_{3,t}, \ldots, x_{m,t}\}$$

- Assume that password y is found at the endpoint indexed j, i.e. $y = x_{j,t}$
- Looking at the chain we see that the hash value $h_{j,t}$ is reduced to password y

$$x_{j,1} \xrightarrow{hash} h_{j,1} \xrightarrow{reduce} x_{j,2} \xrightarrow{hash} \ldots \xrightarrow{reduce} x_{j,t-1} \xrightarrow{hash} \mathbf{h_{j,t-1}} \xrightarrow{reduce} \mathbf{x_{j,t}} = \mathbf{y}$$

- The password $x_{j,t-1}$ hashes to value $h_{j,t-1}$, thus $x_{j,t-1}$ is the correct password!

$$x_{j,1} \xrightarrow{hash} h_{j,1} \xrightarrow{reduce} x_{j,2} \xrightarrow{hash} \ldots \xrightarrow{reduce} \mathbf{x_{j,t-1}} \xrightarrow{hash} \mathbf{h_{j,t}} \xrightarrow{reduce} x_{j,t} = y$$

Rainbow Tables

Attack Step (online)

- **Case 1 (continued).** Note that the rainbow table has stored only the start and end points $(x_{j,1}, x_{j,t})$
- To compute the right password $x_{j,t-1}$ we start at with the start point $x_{j,1}$ which is stored in the rainbow table
- We apply the function $f(\cdot)$ $t-1$ times, reaching the password

$$x_{j,1} \xrightarrow{f} x_{j,2} \xrightarrow{f} x_{j,3} \xrightarrow{f} \dots \xrightarrow{f} x_{j,t-1}$$

- It is possible that the end point $x_{j,t}$ has more than a single preimage. We refer to this as a **false alarm**.

Rainbow Tables

Attack Step (online)

- **Case 2.** The password y is not one of the end points of the rainbow table

$$y \notin \{x_{1,t}, x_{2,t}, x_{3,t}, \ldots, x_{m,t}\}$$

- Then we apply the function $f(\cdot)$ on y and repeat until it matches any of the end points

$$y = f(y) = reduce(hash(y))$$

Rainbow Tables

Attack Step (online)

- **Case 2.** The password y is not one of the end points of the rainbow table

$$y \notin \{x_{1,t}, x_{2,t}, x_{3,t}, \ldots, x_{m,t}\}$$

- Then we apply the function $f(\cdot)$ on y and repeat until it matches any of the end points

$$y = f(y) = reduce(hash(y))$$

- The process is probabilistic

$$\text{probability of success } \geq \frac{1}{n} \sum_{i=1}^{m} \sum_{j=0}^{t-1} \left(1 - \frac{it}{n}\right)^{j+1},$$

where n the total amount of passwords and m, t the rainbow table parameters

- Typically we construct different tables with different reduction functions thus naming the process rainbow tables

Rainbow tables

How to limit password attacks

- **Policy.** Good password policies or system-assigned passwords: enforce long passwords and special characters, notify the user about the password strength, generate random passwords
- **Salting.** Hashing passwords with salt makes precomputed hash tables obsolete and forces the attacker to compute hashes online
- **Iterated hashing.** To slow down an attacker that is hashing passwords, the system can apply the hash multiple times when registering a user i.e. compute $hash^{1000}(x)$ for user password x
- **Special hash functions.** General cryptographic hash functions are fast, use slower ones on purpose such as $Argon2$ and $bcrypt$

One-Time Passwords

OTP

- A major password problem is their static nature
- If somehow recovered, passwords can be easily re-used to establish authentication

OTP

- A major password problem is their static nature
- If somehow recovered, passwords can be easily re-used to establish authentication

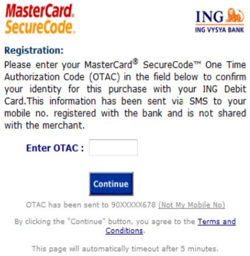
- We can use **one-time passwords (OTP)** that are valid for one use only
- The OTPs are pre-shared between the party that must be authenticated and the verifier that confirms it

OTP

- A major password problem is their static nature
- If somehow recovered, passwords can be easily re-used to establish authentication

- We can use **one-time passwords (OTP)** that are valid for one use only
- The OTPs are pre-shared between the party that must be authenticated and the verifier that confirms it

- How do we share these passwords?
- We generate them on the system and then send them to the user
  e.g. get them in a letter from your bank, through SMS, using a custom device



OTP

Lamport OTP scheme

- We want to avoid transmitting OTPs in the clear thus we can use the **Lamport hash chain** between a user and a system
- The scheme can efficiently generate OTPs using an initial secret w and a hash function $hash(\cdot)$

OTP

Lamport OTP scheme

- We want to avoid transmitting OTPs in the clear thus we can use the **Lamport hash chain** between a user and a system
- The scheme can efficiently generate OTPs using an initial secret w and a hash function $hash(\cdot)$

1. **Setup phase.** The user has a secret w and fixes a constant t
    - The value of t specifies the number of authentications that is allowed e.g. $t = 100$

OTP

Lamport OTP scheme

- We want to avoid transmitting OTPs in the clear thus we can use the **Lamport hash chain** between a user and a system
- The scheme can efficiently generate OTPs using an initial secret w and a hash function $hash(\cdot)$

1. **Setup phase.** The user has a secret w and fixes a constant t
   - The value of t specifies the number of authentications that is allowed e.g. $t = 100$
   - The user applies the hash function t times on the secret w

$$hash^t(w) = hash(hash(\dots hash(w)\dots))$$

OTP

Lamport OTP scheme

- We want to avoid transmitting OTPs in the clear thus we can use the **Lamport hash chain** between a user and a system
- The scheme can efficiently generate OTPs using an initial secret w and a hash function $hash(\cdot)$

1. **Setup phase.** The user has a secret w and fixes a constant t
    - The value of t specifies the number of authentications that is allowed e.g. $t = 100$
    - The user applies the hash function t times on the secret w

$$hash^t(w) = hash(hash(\dots hash(w)\dots))$$

$$h_{100} = w, h_{99} = hash(w), h_{98} = hash^2(w), \dots, h_1 = hash^{99}(w), h_0 = hash^{100}(w)$$

OTP

Lamport OTP scheme

- We want to avoid transmitting OTPs in the clear thus we can use the **Lamport hash chain** between a user and a system
- The scheme can efficiently generate OTPs using an initial secret w and a hash function $hash(\cdot)$

1. **Setup phase.** The user has a secret w and fixes a constant t
   - The value of t specifies the number of authentications that is allowed e.g. $t = 100$
   - The user applies the hash function t times on the secret w

$$hash^t(w) = hash(hash(\dots hash(w)\dots))$$

$$h_{100} = w, h_{99} = hash(w), h_{98} = hash^2(w), \dots, h_1 = hash^{99}(w), h_0 = hash^{100}(w)$$

$$h_{100} \xrightarrow{hash} h_{99} \xrightarrow{hash} h_{98} \xrightarrow{hash} \dots \xrightarrow{hash} h_1 \xrightarrow{hash} h_0$$

Notice the reverse indexing of the hash chain

OTP

Lamport OTP scheme

- We want to avoid transmitting OTPs in the clear thus we can use the **Lamport hash chain** between a user and a system
- The scheme can efficiently generate OTPs using an initial secret w and a hash function $hash(\cdot)$

1. **Setup phase.** The user has a secret w and fixes a constant t
   - The value of t specifies the number of authentications that is allowed e.g. $t = 100$
   - The user applies the hash function t times on the secret w

$$hash^t(w) = hash(hash(\dots hash(w)\dots))$$

$$h_{100} = w, h_{99} = hash(w), h_{98} = hash^2(w), \dots, h_1 = hash^{99}(w), h_0 = hash^{100}(w)$$

$$h_{100} \xrightarrow{hash} h_{99} \xrightarrow{hash} h_{98} \xrightarrow{hash} \dots \xrightarrow{hash} h_1 \xrightarrow{hash} h_0$$

   Notice the reverse indexing of the hash chain
   - The user transfers in a secure manner $h_0 = hash^t(w)$ to the system and both parties initialize a counter $i = 1$

OTP

Lamport OTP scheme

2. **Authentication phase.** The user can access the system using the following hash value as the one-time password

$$h_i = hash^{t-i}(w)$$

OTP

Lamport OTP scheme

2. **Authentication phase.** The user can access the system using the following hash value as the one-time password

$$h_i = hash^{t-i}(w)$$

   - The user computes h_i and sends to the system the triplet $(userid, i, h_i)$

OTP

Lamport OTP scheme

2. **Authentication phase.** The user can access the system using the following hash value as the one-time password

$$h_i = hash^{t-i}(w)$$

   - The user computes h_i and sends to the system the triplet $(userid, i, h_i)$
   - The system keeps track of the counter for every user and checks that the counter i is expected for this *userid*

OTP

Lamport OTP scheme

2. **Authentication phase.** The user can access the system using the following hash value as the one-time password

$$h_i = hash^{t-i}(w)$$

- The user computes h_i and sends to the system the triplet $(userid, i, h_i)$
- The system keeps track of the counter for every user and checks that the counter i is expected for this *userid*
- The system hashes h_i computing $v = hash(h_i)$

OTP

Lamport OTP scheme

2. **Authentication phase.** The user can access the system using the following hash value as the one-time password

$$h_i = hash^{t-i}(w)$$

- The user computes h_i and sends to the system the triplet $(userid, i, h_i)$
- The system keeps track of the counter for every user and checks that the counter i is expected for this *userid*
- The system hashes h_i computing $v = hash(h_i)$
- The system checks that $v == h_{i-1}$ and if so the user is authenticated
- On a successful authentication, the counter i is incremented and the system saves h_i for the next verification

OTP

Lamport OTP scheme

- Let authentication session $i = 20$ out of $t = 100$ available
- The user will compute the hash value h_{20}:

$$h_{20} = hash^{100-20}(w) = hash^{80}(w)$$

OTP

Lamport OTP scheme

- Let authentication session $i = 20$ out of $t = 100$ available
- The user will compute the hash value h_{20}:

$$h_{20} = hash^{100-20}(w) = hash^{80}(w)$$

- If an attacker gets $h_{20} = hash^{80}(w)$ but does not know the secret w, he cannot compute $h_{21} = hash^{79}(w)$ because $h_{20} = hash(h_{21})$ and he cannot invert the hash function

OTP

Lamport OTP scheme

- Let authentication session $i = 20$ out of $t = 100$ available
- The user will compute the hash value h_{20}:

$$h_{20} = hash^{100-20}(w) = hash^{80}(w)$$

- If an attacker gets $h_{20} = hash^{80}(w)$ but does not know the secret w, he cannot compute $h_{21} = hash^{79}(w)$ because $h_{20} = hash(h_{21})$ and he cannot invert the hash function

- Rephrasing, the attacker cannot use authentication session $i = 20$ in order to bypass authentication session $i = 21$ i.e. a past password cannot be used to derive a future password
- This is the reason of indexing the hashes in reverse

OTP

Lamport OTP scheme

- The user will send to the system the following triplet:

$$(userid, 20, h_{20})$$

- The system will check that the user counter $i = 20$ is in agreement with its own

OTP

Lamport OTP scheme

- The user will send to the system the following triplet:

$$(userid, 20, h_{20})$$

- The system will check that the user counter $i = 20$ is in agreement with its own
- The system hashes h_{20} once:

$$v = hash(h_{20})$$

- If the user provided the correct triplet, then the result should be h_{19}. The system has access to all preceding values in the chain.

$$hash(h_{20}) = hash(hash^{80}(w)) = hash^{81}(w) = h_{19}$$

OTP

Lamport OTP scheme

- The user will send to the system the following triplet:

$$(userid, 20, h_{20})$$

- The system will check that the user counter $i = 20$ is in agreement with its own
- The system hashes h_{20} once:

$$v = hash(h_{20})$$

- If the user provided the correct triplet, then the result should be h_{19}. The system has access to all preceding values in the chain.

$$hash(h_{20}) = hash(hash^{80}(w)) = hash^{81}(w) = h_{19}$$

- If the user would provide a wrong h_{20} (i.e. wrong OTP) then it is very unlikely that it matches h_{19} on the system's side

OTP

Lamport OTP scheme

- The user will send to the system the following triplet:

$$(userid, 20, h_{20})$$

- The system will check that the user counter $i = 20$ is in agreement with its own
- The system hashes h_{20} once:

$$v = hash(h_{20})$$

- If the user provided the correct triplet, then the result should be h_{19}. The system has access to all preceding values in the chain.

$$hash(h_{20}) = hash(hash^{80}(w)) = hash^{81}(w) = h_{19}$$

- If the user would provide a wrong h_{20} (i.e. wrong OTP) then it is very unlikely that it matches h_{19} on the system's side
- On a successful authentication both parties increment the counter i to 21 and the system stores h_{20} for comparison during the next authentication session (session 21)

OTP

- A commercial application of OTPs is the **passcode generator**
- The passcode generator holds a user-specific secret and outputs passcodes
- Passcodes are a function of the user secret, timestamps and system challenges

OTP

- A passcode generator is often used in conjunction with a static password, offering in-depth security
- To login now I need *something that I know* and *something that I have* and we refer to this as a **two-factor authentication**

e.g. a password together with a passcode generator
e.g. a password together with a hardware security token
e.g. a password together with mobile authenticator app
e.g. a PIN together with a smartcard

Biometrics

Biometrics

Biometrics is authentication with *something you are*. Biometrics can be very user-friendly but imply additional costs and pose new technical challenges.

Biometrics

Biometrics is authentication with *something you are*. Biometrics can be very user-friendly but imply additional costs and pose new technical challenges.

Desirable properties of biometrics

- Universal: everyone should possess this feature
  e.g. some people do not have fingerprints (adermatoglyphia)
- Distinguishing: we should be able to distinguish with certainty different users
  e.g. iris scans are very precise while walking (gait) recognition is less potent
- Permanent: the physical characteristic should not change over our lifespan
  e.g. facial features may change a lot (glasses, beard, aging)
- Collectable: easy to collect, without harm or huge effort
  e.g. camera-powered smartphones make faces easy to capture
- Reliable, robust, user-friendly
  e.g. face recognition became much less user-friendly after wearing a mask - iPhones degraded its security to make it usable again

Biometrics

Deploying biometrics

- Biometrics can be used for authentication but also for **identification**
- When identifying the goal is to find the subject from a list of many possible subjects
  e.g. DNA from a crime scene is matching a list of known suspects

Biometrics

Deploying biometrics

- Biometrics can be used for authentication but also for **identification**
- When identifying the goal is to find the subject from a list of many possible subjects
  e.g. DNA from a crime scene is matching a list of known suspects

- Biometrics often requires an **enrollment** phase where the subject enters the biometric features to a database
- This is followed by a **recognition** phase that authenticates or identifies a subject
- Biometric recognition implies errors that can be analyzed using **true positive** and **false negative** rates
- Biometrics can be combined with other authentication methods for in-depth security

Biometrics

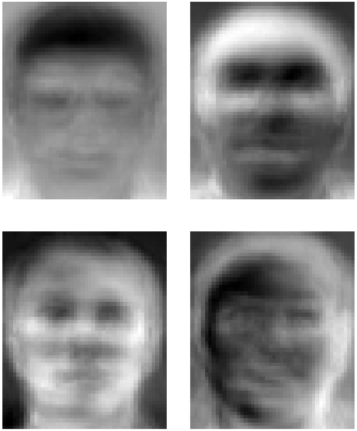
Fingerprints

- Fingerprints have been historically used by forensics investigations
- Distinguishing them requires to extract their special features known as ‘minutia’

Biometrics

Face recognition

- Widely deployed in smartphones and surveillance systems
- Various techniques including 'eigenfaces'



Biometrics

Gait analysis

- Analyzes the motion of the person or animal
- Temporal features and video analysis