Lec 17: User Authentication

CSED415: Computer Security
Spring 2024

Seulbae Kim

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 04 is out!
  - Two weeks (due May 5th)
  - About authentication and entropy (today’s topic)

Overview of CSED415

- First half: About the building blocks of computer security
  - Fundamental security principles and practices
  - Low-level attacks and mitigations
  - Cryptographic primitives

- Second half: About the designs of secure systems
  - How can we develop effective policies to address various security challenges?
  - How can we build secure systems?
  - How can different system components fail?

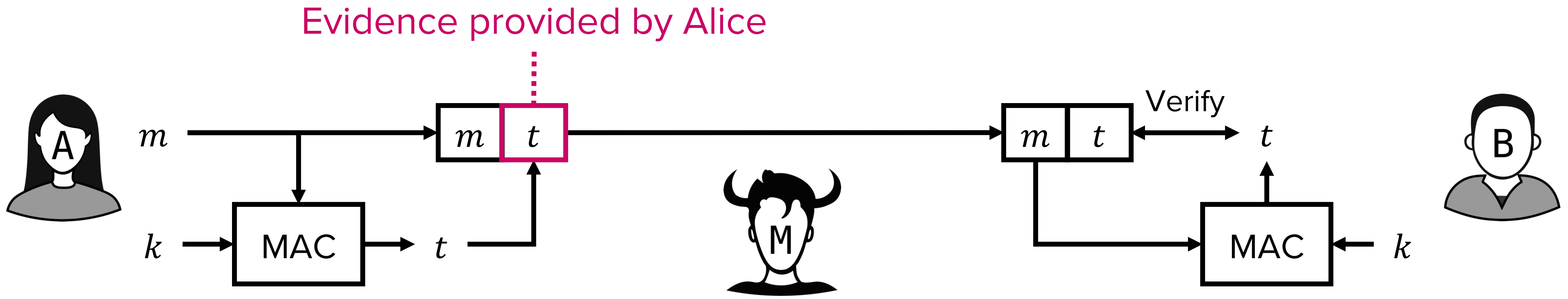
User Authentication

What does “authentication” mean?

- User <u>convincing</u> the system that it is who he or she claims to be during the login process
  - User Alice logs into a system
    - Alice makes a claim, provides an evidence, which is verified by the system
  - After successful login, any process started by her inherits her user identifier (UID)
  - All processes have associated UIDs
  - UID defines on whose behalf a process makes requests

Recall: MAC

- We have already covered a method for authentication
  - Message Authentication Code (MAC)



User authentication in real life

- Photo identification (e.g., driver's license, passport, …)
  - Checking in for flights
  - Taking an exam
  - Pulled over by police
  - Accessing a personal vault
  - Accessing a secure facility
  - …

Your identity is authenticated to ensure that
a service, punishment, or access is granted **only to you**

Why do we need user authentication?

- To establish “Authenticity” and “Accountability” (recall: Lec 02 – CIA+AA)
  - Authenticity
    - Being genuine and being able to be verified and trusted
    - Certifying the integrity of the origin of information
  - Accountability
    - Actions of an entity should be traced back uniquely to the entity
    - A system must be able to trace security breach to the attacker

Three parts of user authentication

- Registration
  - User registers and sets up secret between user and system
- Authentication check
  - User sends (user, secret) along with request
  - System checks if the secret matches its copy
- Recovery
  - User loses the secret
  - Often overlooked

Many challenges exist in each part

Identification (== secret)

- Something unique about you (a user) that a system knows
  - Something you know:
    - Password, Personal Identification Number (PIN), answers to security questions
  - Something you have
    - Smart cards, physical keys, one-time tokens
  - Something you are (static biometrics)
    - Fingerprint, retina, face
  - Something you do
    - Handwriting characteristics (signature), typing pattern

Means of authentication

- Password-based
- Challenge-response
- Biometric
- Zero-knowledge
- Multi-factor

Password-based Authentication

Password-based authentication

- The most widely-used authentication method
- Using “what you know” for authentication
  - You choose and register your password
- The service must store your password somewhere!

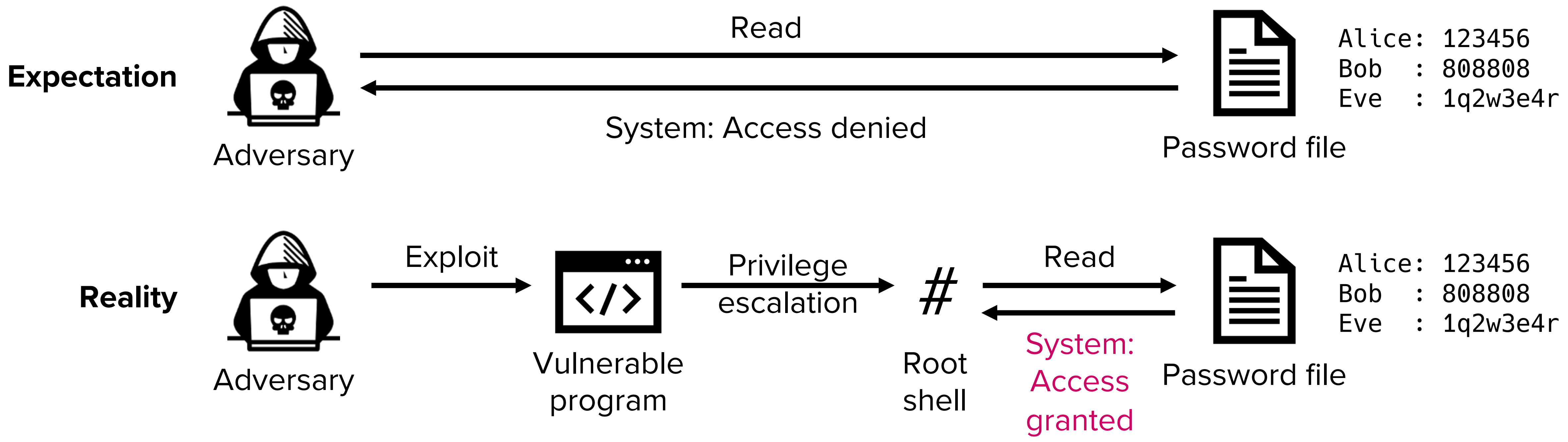
Storing Password

Naïve implementation

- Registration
  - A user registers with a **(username, password)** pair
  - System stores the pair in a system file
- Authentication
  - A user provides an identifier (e.g., username) with a password
  - The system compares the password to a previously registered password
  - Grant access if the passwords are identical

Naïve implementation

- Problem?
  - If attacker compromises the system, he/she can recover all user/password pairs

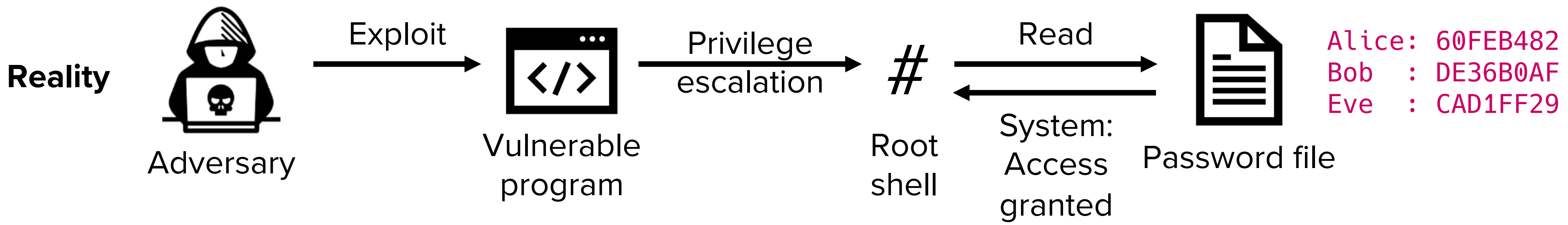


Improvement: Hashed password

- Registration
  - A user registers with a (username, password) pair
  - System stores a **(username, hash(password))** pair in a system file
- Authentication
  - A user provides an identifier (e.g., username) with a password
  - The system computes the hash of the password and compares it to the previously registered **hash(password)**
  - Grant access if identical

Improvement: Hashed password

- Hashed passwords are safer



- Advantage
  - Cryptographic hash functions are one-way, so plaintext passwords will not be easily restored
- Problems?

Improvement: Hashed password

- Problem: Skewed distribution (user factor)
  - Recent password statistics
    - The most common password is “123456” (Readers’ Digest, 2023)
    - 20% of users include their birth year in their password (security.org, 2023)
    - In 2022 alone, over 24 billion passwords were exposed by hackers
      - Of the 24 billion (!!) compromised credentials, only 6.7 billion were unique pairs of username and password (Digital Shadows, 2022)
    - Individuals reuse passwords on 10 of their personal accounts (Ponemon Institute, 2020)
  - Check: Top 10000 passwords (OWASP SecLists)

Improvement: Hashed password

- Problem: Skewed distribution (user factor)
  - Attackers can build “rainbow tables”:
    - Table of password-to-hash mappings
    - Contain hash of possible passwords
      - The most common passwords (e.g., top 10000) are included
    - Expensive to compute, but allows the adversaries to efficiently invert hashes afterwards
  - Many practical hash functions are optimized for performance
    - Rather helps attackers build rainbow tables
    - Using slow hash could slow down attackers, but it does not solve the fundamental problem

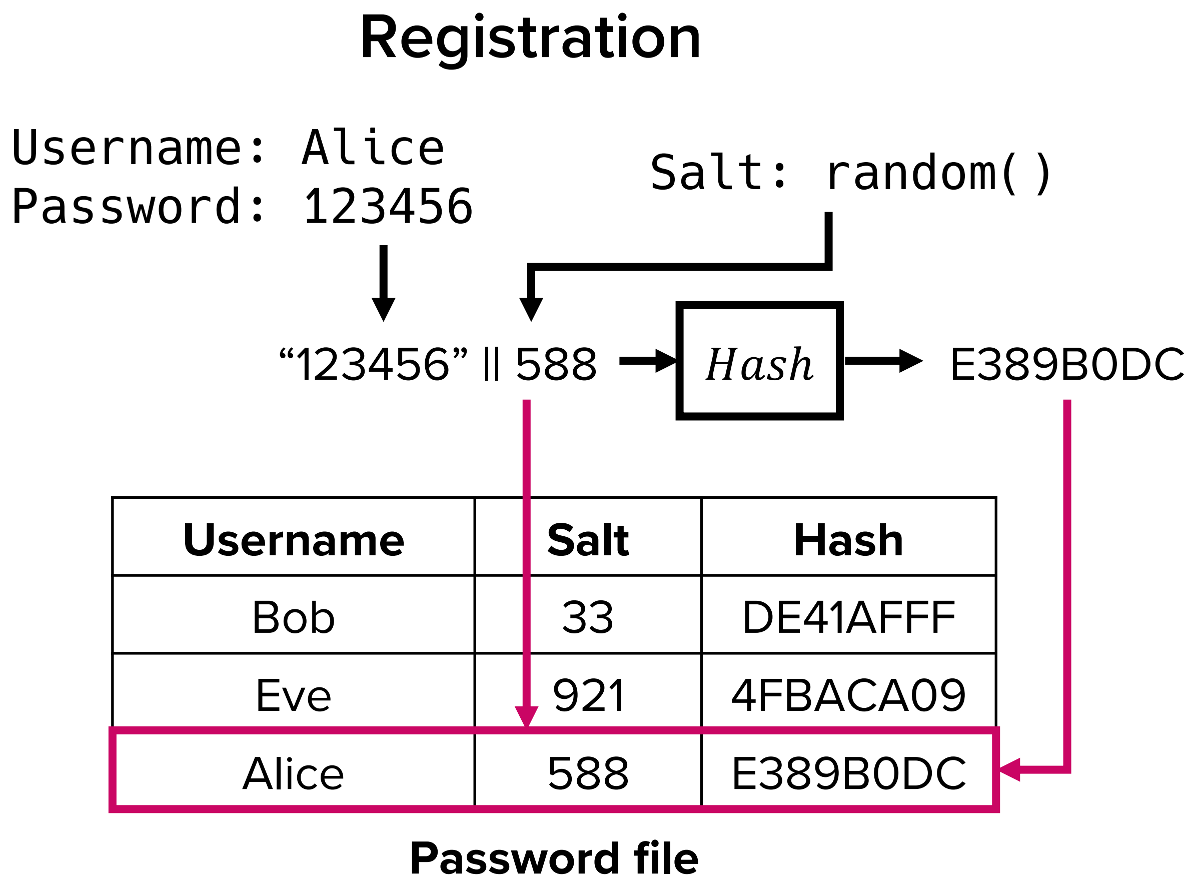
Better solution: Password salting

POSTECH

- Registration
  - A user registers with a (username, password) pair
  - System stores a **(username, hash(password||salt))** pair in a system file
    - **Input a randomness into the password hash**
- Authentication
  - A user provides an identifier (e.g., username) with a password
  - The system computes the hash of (password+salt) and compares it to the previously registered hash(password+salt)
  - Grant access if identical

Better solution: Password salting

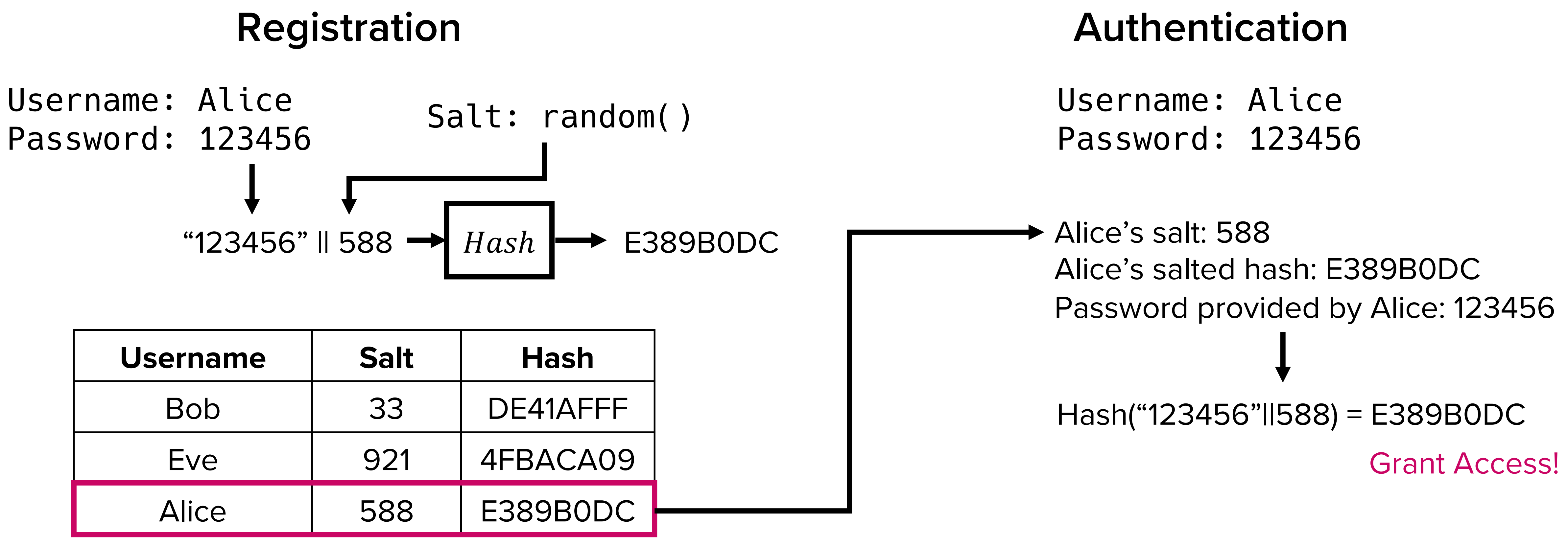
- Salt: A random nonce (i.e., number that is used only once)



Username	Salt	Hash
Bob	33	DE41AFFF
Eve	921	4FBACA09
Alice	588	E389B0DC

Better solution: Password salting

- Salt: A random nonce (i.e., number that is used only once)



Username	Salt	Hash
Bob	33	DE41AFFF
Eve	921	4FBACA09
Alice	588	E389B0DC

Password file

Better solution: Password salting

- Q) Why is it better if the adversary can still compromise the server and obtain the salt?

The adversary cannot use the same rainbow table
(same password now have different hash values due to salts)

It is infeasible to generate rainbow tables for all possible salts
(e.g., using a 16-bit salt → 65,536 possible salts x 10000 passwords)

Better solution: Password salting

- Best practices
  - Choose a long random salt
    - Having more entropy is important
  - Choose a fresh salt every time a user changes password
    - Keeping entropy high is also important

Password-based authentication in practice

- Linux password system
  - Password file: `/etc/shadow`
    - Owner: root
    - Group: shadow (special group with no user)
    - Permission: -rw-r----
      - Only root can read and write
      - Shadow group can read

Password-based authentication in practice

- Linux password system
  - Password file: `/etc/shadow`
    - Entry format:
      `USERNAME:PASSWORD:LASTCHANGE:MIN:MAX:WARNING:INACTIVE:EXPIRE:BLANK`
    - e.g., lab01 account entry:

```
lab01:$y$j9T$zSJOm6PdwWhP1Mvr8gPKZ/$FTvikjg3e63Qe3.84aVQAFndYeKugjrzdXjMdCIuv9A:
19769:0:99999:7:::
```

    - `PASSWORD` field breakdown: `$algorithm$param$salt$saltedhash`
      - `Algorithm: y (yescrypt)`
      - `Param: j9T`
      - `Salt: zSJOm6PdwWhP1Mvr8gPKZ/`
      - `Salted hash: FTvikjg3e63Qe3.84aVQAFndYeKugjrzdXjMdCIuv9A`

Password-based authentication in practice

- Linux password system
  - Password file: `/etc/shadow`
    - Entry format:
      `USERNAME:PASSWORD:LASTCHANGE:MIN:MAX:WARNING:INACTIVE:EXPIRE:BLANK`
    - e.g., test account entry:

```
test:$5$HSNRch0WVDo2P271$K8IIkRrc7LKyhkHzvQXGoDxdhnC8xa2WhrPkEdmyulC:
19769:0:99999:7:::
```

    - PASSWORD field breakdown: `$algorithm~~$param~~$salt$saltedhash`
      - `Algorithm: 5 (SHA-256)`
      - `Param: None // Not all algorithms use parameters`
      - `Salt: HSNRch0WVDo2P271`
      - `Salted hash: K8IIkRrc7LKyhkHzvQXGoDxdhnC8xa2WhrPkEdmyulC`

Password-based authentication in practice

- Linux password system
  - Password file: `/etc/shadow`
    - Verify lab01's password through a Perl one-liner:

```
$ perl -e 'print crypt("EZJUxyuF", "\$y\$j9T\$zSJOm6PdwWhP1Mvr8gPKZ/\$")'
```

lab01's password　　　　Salt from /etc/shadow

    - Verify test's password through OpenSSL:

```
$ openssl passwd -5 -salt HSNRch0WVDo2P271 1q2w3e4r
```

Salt from /etc/shadow　　test's password

The results match the salted hash in /etc/shadow

Password-based authentication in practice

- Linux password system
  - Q) What can you do with a leaked `/etc/shadow` file?

```
lab01:$y$j9T$zSJOm6PdwWhP1Mvr8gPKZ/$FTvikjg3e63Qe3.84aVQAFndYeKugjrzdXjMdCIuv9A:
19769:0:99999:7:::
```

```
test:$5$HSNRch0WVDo2P271$K8IIkRrc7LKyhkHzvQXGoDxdhnC8xa2WhrPkEdmyulC:
19769:0:99999:7:::
```

Password-based authentication in practice

- Linux password system
  - Refer to "`man 5 shadow`" and "`man 3 crypt`" for more details!

Selecting Good Password

Selecting "strong" passwords?

- Common password format requirements:
  - Contain both lowercase and uppercase letters
  - Contain at least one digit (0-9)
  - Contain at least one special character
  - Some special characters are not allowed (#, $, …)
  - …

Do the format requirements really help?
e.g., is "password1!" strong?

Selecting “strong” passwords?

- Attackers can model the password constraints and generate rainbow tables
  - Each requirement provide directions for generating the table “Generate a rainbow table with 8+ chars, having both uppercase and lowercase, having at least one symbol but no ‘#’, …”

- Format requirements do not translate to better password security
  - What matter more are “password entropy” and “password reuse”

Towards strong passwords: Password entropy

- Password entropy
  - Definition: Probability that someone picks a specific password
  - Expressed in terms of of bits
    - Entropy of a known password: 0 bit
    - If password can be guessed on the first attempt half the time: 1 bit
    - Password of 16 bits of entropy requires 2^{16} guesses
  - Examples (estimating entropy)
    - "`password`": 0 bit (known password)
    - "`password1`": $\log_2 10 = 2.3$ bits (known + one digit)
    - "`pa$$w0rd`": Higher, but not enough (s to $ substitution is easily modeled)
    - "`85a60bed`": Very high

Towards strong passwords: Password entropy

POSTECH

- Password entropy
  - Good practice: Choose random (high entropy) passwords
  - Problem: User factor
    - Users are tempted to use simple, easy-to-remember passwords
    - End up picking low entropy passwords. Bad idea!

  - Solution: Password manager
    - Picks high entropy passwords and securely stores them

    → Strong password w/o efforts to remember

Towards strong passwords: Password reuse

POSTECH

- Password reuse
  - Definition: Using the same password across different systems
  - Why is it a bad idea?
    - Different systems have different levels of security
      - Google “might” keep your password safer
      - Discord, Reddit, Bank of America, … fell victim to data breach in 2023
        - Google’s strong security does not matter if your password is leaked elsewhere
      - Sketchy services (e.g., porn sites) even sell their password databases

    → Weakest link matters!
      - One leak compromises all of your accounts if you reuse a password

Towards strong passwords: Password reuse

- Password reuse
  - Good practice: Never reuse the same password across different services
  - Problem: User factor (again!)
    - Users are tempted to reuse the same password for convenience
  - Solution: Password manager (again!)
    - Picks different, strong password for different services

    → Different password

Towards strong passwords

- Password manager
  - Selects high-entropy passwords
  - Selects different passwords for different services
  - Securely stores the passwords
  - User must authenticate to password manager
    - Must remember one strong password
    - Much better than generating n strong passwords for n different services and remembering them all

Remaining problem: Brute-force attacks

- Our previous approaches
  - Securely storing password with salt and hashing
  - Selecting high-entropy password
- Still vulnerable to brute-force attacks
  - No matter how well the passwords are chosen and stored, attacker can still try all possible passwords until he/she succeeds

How can we prevent guessing/brute-force attacks?

Preventing Brute-forcing

Preventing brute-force attacks

- Rate-limiting
  - Rate-limit the number of guesses
  - Implement time-out periods after too many incorrect guesses

- Deliberately slowing down authentication
  - Using slow hash functions or iterating a hash function
    - Slow verification throttles guessing speed
    - Generation of rainbow table is also slowed down

Password Recovery

Password recovery

- Very important but often overlooked
  - Password entropy is important (obvious) and well taken care of
  - What about recovery entropy?
    - Recall: Sarah Palin email hack incident (Lec 01)
      - If a user forgets the password, Yahoo allowed logging in by answering security questions
      - Sarah Palin’s security question asked her birthday
      - She was a VP candidate for US presidential election. Her birthday was on Wikipedia
      - == Zero entropy!
      - Even if she was not famous, the entropy is only $\log_2 365 = 5.89$ bits
    - Answers to security questions inherently possess low entropy
      - “Make of your first car”, “Your mother’s maiden name”, “The city of your birth”, …

Password recovery

- Very important but often overlooked
  - Formally, the strength of a password authentication scheme is `min(password_entropy, recovery_q_entropy)`

- Good security practice
  - Never let users log in through recovery questions
  - Use another factor for recovery authentication
    - e.g., rend recovery link to email address, phone number
  - Users with the recovery link can reset the password and then login

“Your Pa$$word Doesn’t Matter”

Your Pa$$word doesn't matter

- A blog article from Microsoft
  - Claim: Passwords are broken
    - Supports the claim by showing how passwords are vulnerable to major attacks
      - Credential stuffing
      - Phishing
      - Keylogging
      - Local discovery
      - Extortion
      - Spraying
      - Brute-forcing

Your Pa$$word doesn't matter

- What do we do then?
  - Need other methods on top of passwords

  → Next lecture!

Coming up next: More authentication methods

- Password-based ✅
- Challenge-response
- Biometric
- Zero-knowledge
- Multi-factor

Questions?