

Lec 15: User Authentication

CSED415: Computer Security
Spring 2025

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 04 has been released!
 - About password-based authentication and entropy (today's topic)
 - Due on April 25

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 different system components fail?
 - How can we build secure systems and test their security?

User Authentication

What does “authentication” mean?

- Authentication: User **convincing** the system that he or she is the claimed identity during the login process
 - Example (Linux):
 - User Alice wants to log into a system
 - Alice makes a claim that she is indeed Alice by entering her username “alice”
 - Alice provides evidence, e.g., her password
 - The system verifies that the password is correct for user “alice”

Real-world user authentication

- Photo identification (e.g., driver's license, passport, ...)
 - Checking in for flights
 - Taking an exam
 - Pulled over by the 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 ensure **A**uthenticity and **A**ccountability (recall: Lecture 02 – CIA+**AA**)
 - With authenticity, we can
 - Verify and trust the identity
 - Certify the integrity of the origin of information
 - With accountability, we can
 - Trace actions back to a specific entity
 - e.g., If a security breach happens, the system can determine who is responsible

Three parts of user authentication

- Registration
 - User registers and sets up a secret with the system
- Authentication check
 - The user sends a <username, secret> pair with the login request
 - The system checks if the user's secret matches the secret it stored
- Recovery
 - The user loses the secret and needs a way to regain access
 - This part is often overlooked but crucial for a secure design

Identification (== Secret)

- Something **unique** about you that the system can verify
 - Something you (but no one else) know:
 - Password, Personal Identification Number (PIN), answers to security questions
 - Something you (but no one else) have:
 - Smart cards, physical keys, one-time tokens, etc.
 - Something you (but no one else) are:
 - Static biometrics (fingerprint, retina, facial features, etc.)
 - Something you (but no one else) do:
 - Handwriting characteristics (e.g., signature), typing pattern, etc.

Means of authentication

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

Password-based Authentication

Password-based authentication

- The most widely used authentication method
- Uses “what you know” for authentication
 - You choose and register your password
 - No one else knows your password
- The service must **store** your password somewhere to verify it when you log in

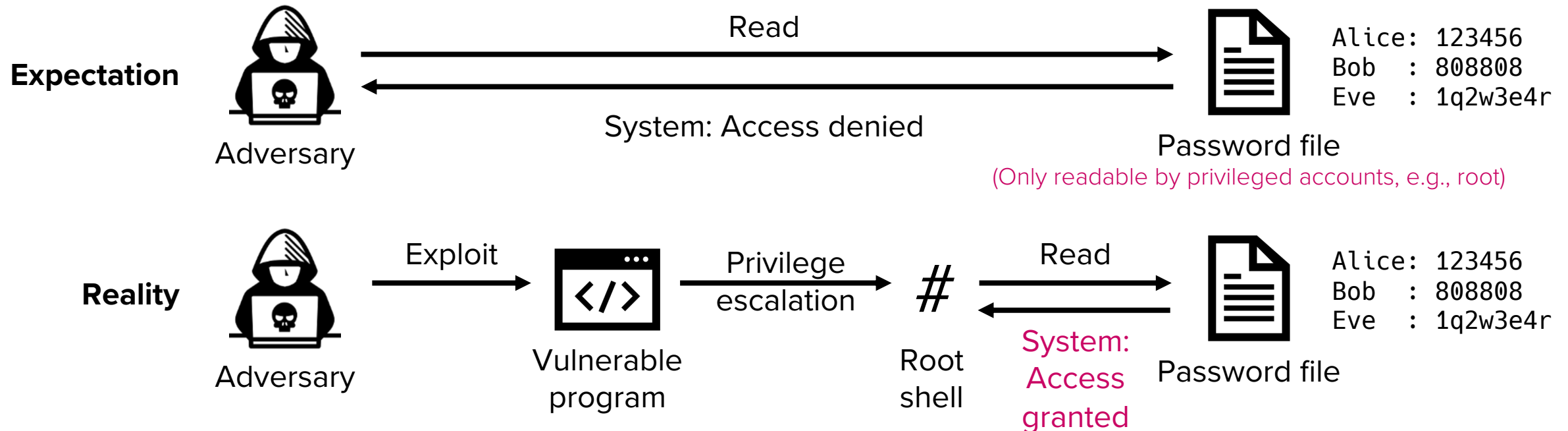
Storing Password

Naïve implementation

- Registration
 - A user registers with a <username, password> pair
 - The system stores the pair in a system file
- Authentication
 - The user provides an identifier (e.g., username) with a password
 - The system compares the provided password to the previously registered password
 - If they match, grant access

Naïve implementation

- Problem?
 - If an attacker compromises the system, the attacker can recover all stored <username, password> pairs

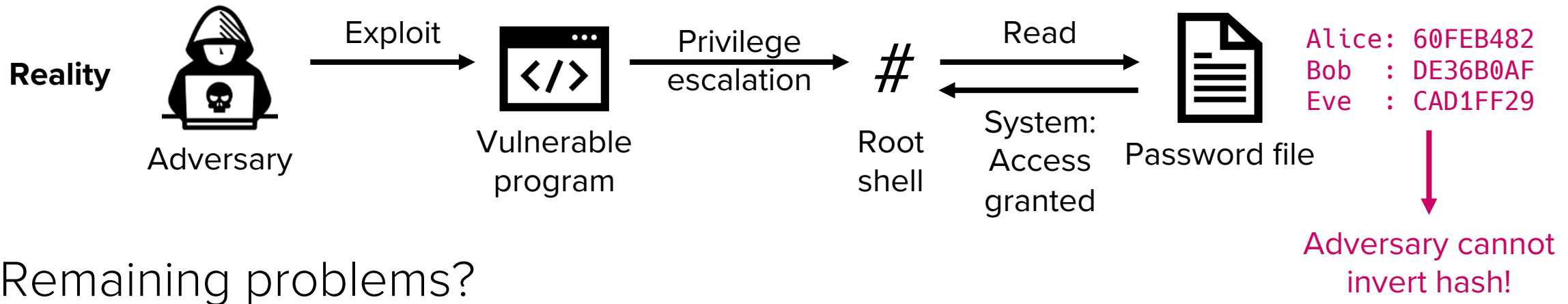


Improvement: Hashed password

- Registration
 - A user registers with a <username, password> pair (same as before)
 - The system stores <username, hash(password)> in a system file
- Authentication
 - The 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
 - Cryptographic hash functions are one-way, so plaintext passwords are not easily restored



- Remaining problems?

Improvement: Hashed password

- Problem: Skewed distribution (user factor)
 - Recent password-related 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
 - 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 out [Top 10000 passwords](#) (OWASP SecLists) for a sense of how predictable many user-chosen passwords are

Improvement: Hashed password

- Problem: Skewed distribution (user factor)
 - Attackers can build “rainbow tables”:
 - A precomputed mapping of common passwords to their hashes
 - Attackers can include the most common passwords (e.g., top 10000)
 - They can also include the permutations of allowed letters
 - Expensive to compute, but allows the adversaries to efficiently invert hashes afterwards
 - Many practical hash functions are optimized for speed
 - 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

- Registration

- A user registers with a <username, password> pair
- The system generates a random salt for the user
- Then it stores a <username, hash(password ⊗ salt)> pair and the salt
 - ⊗ can be any operation (sum, string concatenation, xor, ...)

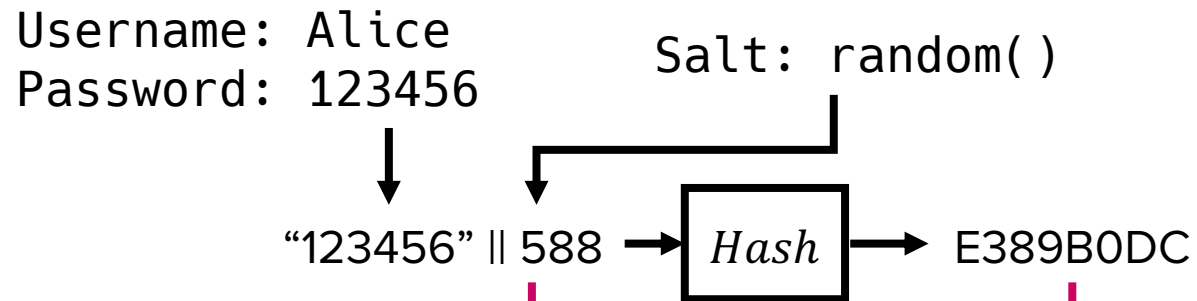
- Authentication

- The user provides an identifier (e.g., username) with a password
- The system retrieves the salt for that user and computes hash(password ⊗ salt)
- If it matches the stored salted hash, grant access

Better solution: Password salting

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

Registration



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

Password file

Better solution: Password salting

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

Registration

Username: Alice
Password: 123456

Salt: random()

"123456" || 588 →

Hash

E389B0DC

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

Password file

Authentication

Username: Alice
Password: 123456

Alice's salt: 588
Alice's salted hash: E389B0DC
Password provided by Alice: 123456

hash("123456" || 588) = E389B0DC

Match. Grant Access!

Better solution: Password salting

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

The adversary cannot use the same precomputed 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 per password)

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 the password
 - Keeping entropy high is also important

Password-based authentication in practice

- Linux stores passwords in `/etc/shadow`
 - Owner: **root** (the most privileged admin account)
 - Group: **shadow** (special group with no user)
 - Permission: -rw-r-----
 - Only the **root** account can read and write
 - Members of group **shadow** can read
 - Regular user accounts cannot read or write to the file

(We will discuss the permission system in more detail next week)

Password-based authentication in practice

- Linux stores passwords in /etc/shadow
 - Entry format:
USERNAME:PASSWORD:LASTCHANGE:MIN:MAX:WARNING:INACTIVE:EXPIRE:BLANK
 - e.g., entry for csed415-lab01 account:

```
csed415-lab01:$y$j9T$pBv0gISfSWc8q9L5ZXy0f0$GILm0bDnlQGhEf0VezeWSofl.Upv1ycdjd2l2ITugP6:20137:0:99999:7:::
```

- PASSWORD field breakdown: \$algorithm\$param\$salt\$saltedhash
 - Algorithm: **y** (yescrypt)
 - Param: **j9T** (configuration parameter for yescrypt encryption)
 - Salt: **pBv0gISfSWc8q9L5ZXy0f0**
 - Salted hash: **GILm0bDnlQGhEf0VezeWSofl.Upv1ycdjd2l2ITugP6**

Password-based authentication in practice

- Linux stores passwords in /etc/shadow
 - Entry format:
USERNAME:PASSWORD:LASTCHANGE:MIN:MAX:WARNING:INACTIVE:EXPIRE:BLANK
 - e.g., entry for testuser account:

```
testuser:$5$HSNRch0WVDo2P271$K8IIkRrc7LKyhkHqvQXGoDxdhnC8xa2WhrPkEdmyu1C:20188:0:99999:7:::
```

- PASSWORD field breakdown: ~~\$algorithm\$param~~\$salt\$saltedhash
 - Algorithm: 5 (SHA-256)
 - Param: None (SHA-256 does not require config parameter)
 - Salt: HSNRch0WVDo2P271
 - Salted hash: K8IIkRrc7LKyhkHqvQXGoDxdhnC8xa2WhrPkEdmyu1C

Password-based authentication in practice

- Linux stores passwords in /etc/shadow
 - Verifying csed415-lab01's password through a Perl one-liner:

```
$ perl -e 'print crypt("2aea0b7b", "\$y\$j9T\$pBv0gISfSWc8q9L5ZXy0f0\$")'
```

csed415-lab01's password

Salt of csed415-lab01 found in /etc/shadow

- Verifying testuser's password through OpenSSL:

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

Salt found in /etc/shadow Password of testuser

The results match the entries in /etc/shadow

Password-based authentication in practice

- Linux stores passwords in /etc/shadow
 - Q) What can you do with a leaked /etc/shadow file?

```
csed415-lab01:$y$j9T$pBv0gISfSWc8q9L5ZXy0f0$GILm0bDnlQGhEf0VezeWSofl.Upv1ycdjd2l2ITugP6:20137:0:99999:7:::
```

```
testuser:$5$HSNRch0WVDo2P271$K8IIkRrc7LKyhkHqvQXGoDxdhnC8xa2WhrPkEdmyulC:20188:0:99999:7:::
```

- A) You know the hash function, params, salt, and the salted hash
 - You can attempt to brute-force the password by generating potential passwords, salting them, and hashing to see if the result matches

Password-based authentication in practice

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

Selecting Good Passwords

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?
For example, is “Password1!” strong?

Selecting “strong” passwords?

- Attackers can model the password constraints to generate rainbow tables
 - Each requirement provide directions for generating the table
“Generate a rainbow table with 8+ chars, having both uppercase and lowercase letters, having at least one symbol but no ‘#’, ...”
- In fact, format requirements do not lead to better password security
 - What matter more are “using high-entropy passwords” and “avoiding password reuse”

Towards strong passwords: Password entropy

- Use high-entropy passwords
 - Entropy: Probability that someone selects a specific password
 - Password entropy is expressed in terms of bits
 - Entropy of a known password: 0 bit
 - If password can be guessed on the first attempt with 50% chance: 1 bit
 - A password with 16-bit entropy requires 2^{16} guesses to be found
 - Entropy examples
 - “password”: 0 bit (known password)
 - “password1”: $\log_2 10 = 2.3$ bits (known password + one digit (0-9) to guess)
 - “pa\$\$w0rd”: Higher, but not enough (s to \$ substitution is easily modeled)
 - “85a60bed”: Fairly high

Towards strong passwords: Password entropy

- Use high-entropy passwords
 - Good practice: Choose random (high-entropy) passwords
 - Problem: User factor
 - Users are tempted to use simple, easy-to-remember passwords
 - People usually end up selecting low entropy passwords
- Systematic solution: Password manager
 - Password manager generates high-entropy passwords for you
 - It then securely stores them
 - Help users use strong passwords without efforts to memorize them

Towards strong passwords: Password reuse

- Avoid password reuse
 - Password reuse: Using the same password across multiple services
 - 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
 - Remember, weakest link matters!
 - One leak compromises all your accounts if you reuse a password

Towards strong passwords: Password reuse

- Avoid 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
 - Systematic Solution: Password manager (again!)
 - Generates unique passwords for different services

Towards strong passwords

- Password manager
 - Selects high-entropy passwords
 - Selects different passwords for different services
 - Securely stores the passwords
 - To access the stored passwords, user must authenticate to the password manager
 - User must generate and remember **one** strong password for the password manager
 - Still, it is much easier than generating n strong passwords for n different services and remembering them all

Remaining problem: Brute-force attacks

- Our previous approaches
 - Securely storing passwords as salted hashes
 - Selecting high-entropy passwords
- Problem: Still vulnerable to brute-force attacks
 - No matter how well the passwords are chosen and stored, attacker can still attempt to login with all possible passwords until he/she succeeds

How can we prevent guessing/brute-force attacks?

Preventing Brute-forcing

Methods to prevent brute-force attacks

- Rate-limiting
 - Rate-limit the number of guesses within a given time
 - Temporarily lock accounts after several failed attempts



Methods to prevent brute-force attacks

- Deliberately slowing down authentication
 - Use slow, memory-hard hash functions
 - e.g., bcrypt, yescript, etc.
 - Slow hash verification throttles authentication speed
 - Generation of rainbow table is also slowed down

Both rate-limiting and slow hashes are simple yet effective solutions for brute-force attacks

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 (Lecture 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
 - Entropy: 0 bit
 - Even if she was not someone famous, the entropy is only $\log_2 365 = 5.89$ bits
 - Answers to most 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:
$$\min(\textit{entropy}_{\textit{password}}, \textit{entropy}_{\textit{recovery}})$$
- Good security practice
 - Never let users log in directly through recovery questions
 - Instead, use another factor for recovery authentication
 - e.g., send recovery link to user's email address or 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


- Read: [A blog article from Microsoft](#)
 - Claim: Passwords are broken as they are vulnerable to major attacks
 - Credential stuffing (e.g., purchasing leaked passwords) → Your exact passwords are known
 - Phishing → You give away your password to the attacker
 - Keylogging → You type in your password
 - Local discovery (e.g., dumpster diving) → Exact passwords are discovered
 - Extortion (e.g., threatening) → You give away your password
 - Spraying and brute-forcing → The attacker will eventually succeed

Your Pa\$\$word doesn't matter

- What do we do then?
 - We need other methods on top of passwords
 - Next lecture!

Coming up next: More authentication methods

POSTECH

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

Questions?