Cryptography Engineering

- Lecture 10 (Jan 14, 2026)

- Today's notes:
  - Password and password security
  - Salting, and use TLS to protect passwords

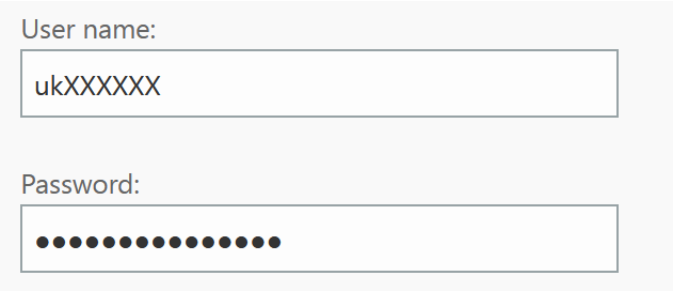
Password and its Security

- **Password:**

  “admin” “123456” “[Your_Name][Your_Birthday]”

  “root” “b8sdhazyn216fsgk.]02=2v4h”

- **Widely used in practice**



Password and its Security

- **Why password security is important:**

Within a couple of weeks, however, Adobe was forced to acknowledge that a more accurate figure for the number of people who were impacted by the hack was some 38 million active users after **a 3.8GB file containing more than 150 million usernames/passwords was dumped on the net.** 5 Nov 2013

Facebook Stored Hundreds of Millions of User Passwords ...

21 Mar 2019 — **Hundreds of millions of Facebook users** had their account passwords stored in plain text and searchable by thousands of Facebook employees ...

LinkedIn
https://www.linkedin.com › news › story › nearly-10-bill...

Nearly 10 billion passwords leaked

In a **leak** that cybersecurity researchers are calling the largest of all time, almost 10 billion unique passwords have been posted to a hacking forum.

The Biggest Password Leak in History

In an unprecedented cyber security event, the largest **password leak** ever recorded has just occurred, exposing over 10 billion passwords.

At the end of 2010, an incident that is known as CSDN Password Leakage Incident happened, and passwords from five websites, including CSDN, Tianya, Duduniu, 7k7k and 178.com, were leaked in several consecutive days. The total number of leaked accounts is over 80 million, and all the leaked passwords are in plaintext. 20 Aug 2014

(source: Google search)

Password and its Security

- **Properties of passwords:**
  - Mainly used for authentication (e.g., hash and compare), easy to replace,…
  - Human-generated and memorizable
  - **Short length, Low Entropy**
  - Highly **Reused**
  - …

Password and its Security

- **Low Entropy**
  - Lack of randomness, predictability, Short length, Limited character set,…
  - Example: (Most people use their personal email as website accounts, e.g., Amazon, …)

    Account: “[YourName]@gmail.com”

    “admin”, “123456”, “hello123”, …

    “[Your/Your Partner’s Name]_[Birthday]”, …

    “[Your Phone number]”, “[Family’s phone number]”…

    “qwerty” (English keyboard), “qwertz” (German keyboard), …

Password and its Security

- **Low Entropy**
  - Lack of randomness, predictability, Short length, Limited character set,…
  - Example: (Most people use their personal email as website accounts, e.g., Amazon, …)

    Account: “[YourName]@gmail.com”

    “admin”, “123456”, “hello123”, …

    “[Your/Your Partner’s Name]_[Birthday]”, …

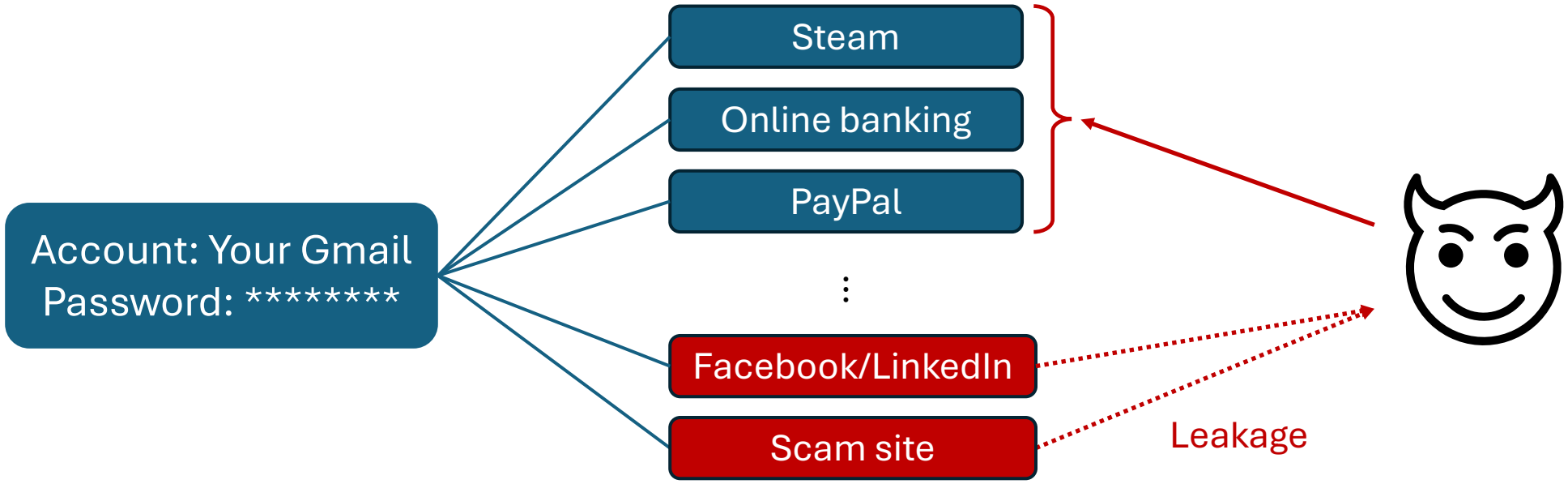
    “[Your Phone number]”, “[Family’s phone number]”…

    “qwerty” (English keyboard), “qwertz” (German keyboard), …

  - **Short, patterned, no randomness, and highly related to personal information**

Password and its Security

- **Highly reused:** Different portals, but the same password…



Password Dictionary

- **Dictionary Attack:**
  - Attack (Guess) using password dictionaries
  - Focus on known/common password combinations, more efficient than brute force…

123456
password
12345678
qwerty
123456789
12345
1234
111111
1234567
dragon
123123
baseball
abc123
football
monkey
letmein
696969
shadow
master
666666
qwertyuiop
123321
mustang
1234567890
michael
654321
pussy
superman
1qaz2wsx

nascar
monster
tigers
yellow
xxxxxx
123123123
gateway
marina
diablo
bulldog
qwer1234
compaq
purple
hardcore
banana
junior
hannah
123654
porsche
lakers
iceman
money
cowboys
987654
london
tennis
999999
ncc1701
coffee
scooby
0000
miller
boston
q1w2e3r4
fuckoff
brandon
yamaha
chester

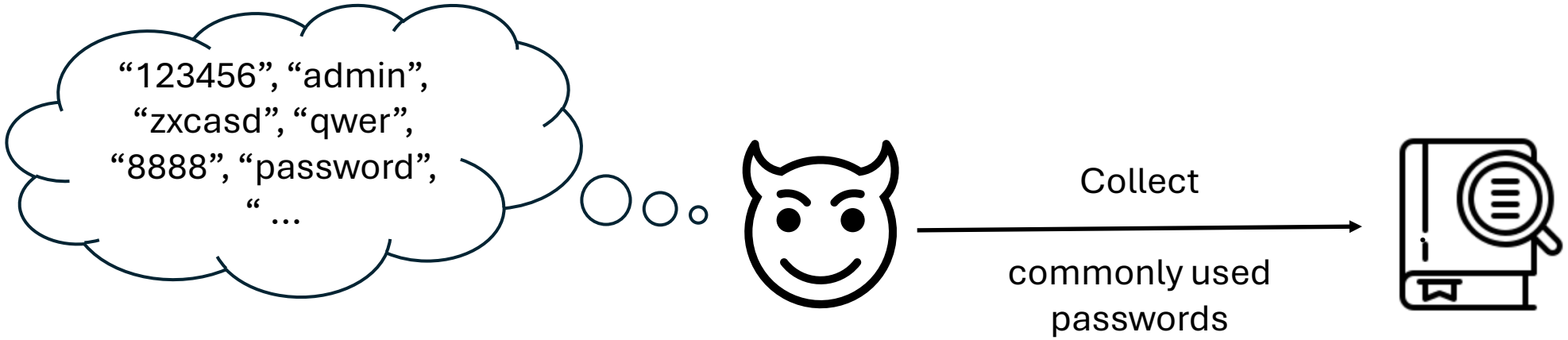
abcd1234
scorpion
qazwsxed
101010
butter
carlos
password
dennis
slipknot
qwerty12
booger
asdf
1991
black
startrek
12341234
cameron
newyork
rainbow
nathan
john
1992
rocket
viking
redskins
butthead
asdfghjk
1212
sierra
peaches
gemini

braves
shelby
godzilla
beaver
fred
tomcat
august
buddy
airborne
1993
1988
lifehack
qqqqqq
brooklyn
animal
platinum
phantom
online
xavier
darkness
blink182
power
fish
green
789456123
voyager
police
travis
12qwaszx
heaven
snowball
lover

bond007
alexis
1111111
samson
5150
willie
scorpio
bonnie
gators
benjamin
voodoo
driver
dexter
2112
jason
calvin
freddy
212121
creative
12345a
sydney
rush2112
1989
asdfghjk
red123
bubba
4815162342
passw0rd
trouble
gunner
happy

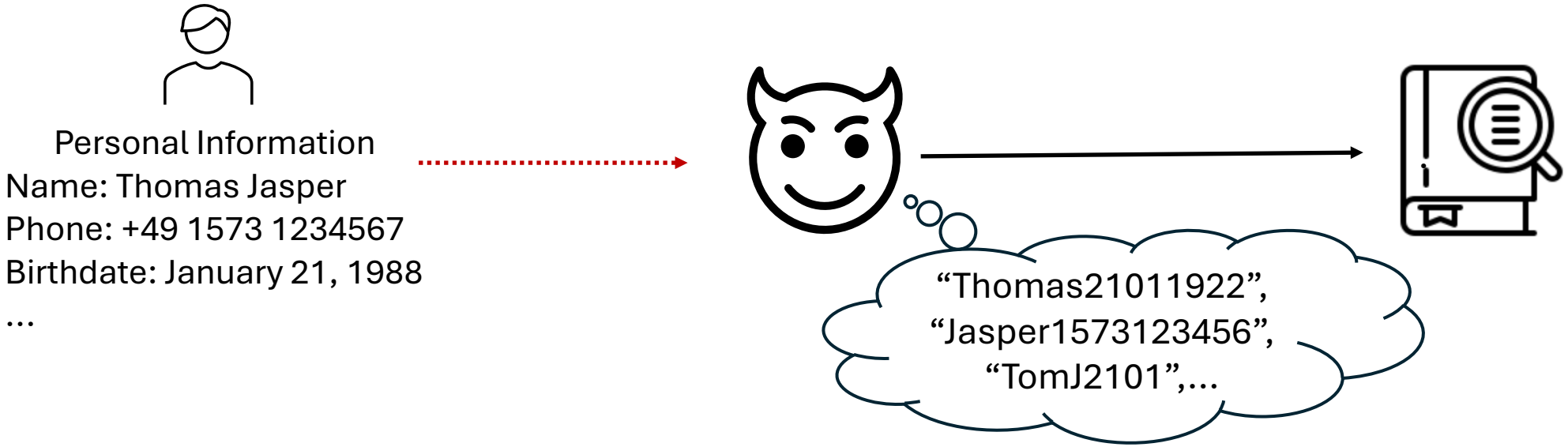
Password Dictionary

- Construct a password dictionary:



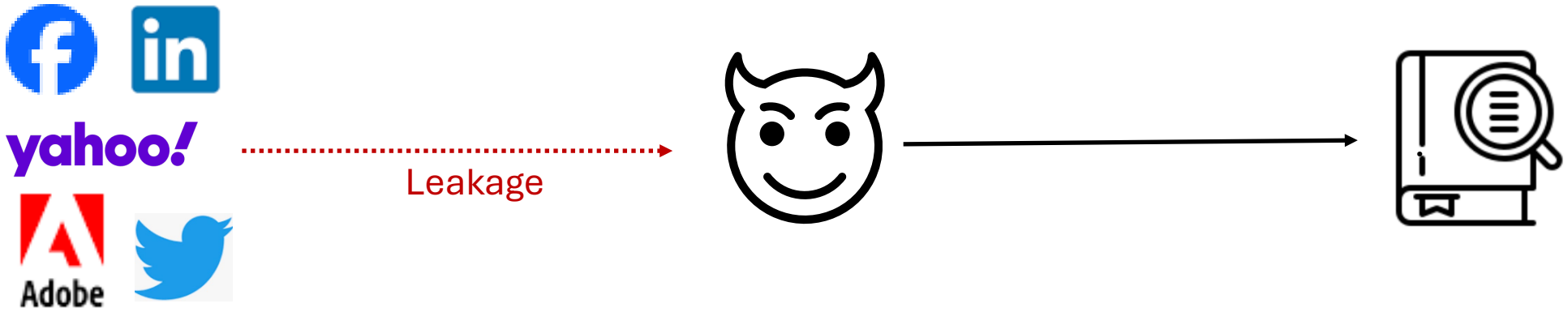
Password Dictionary

- Construct a password dictionary:



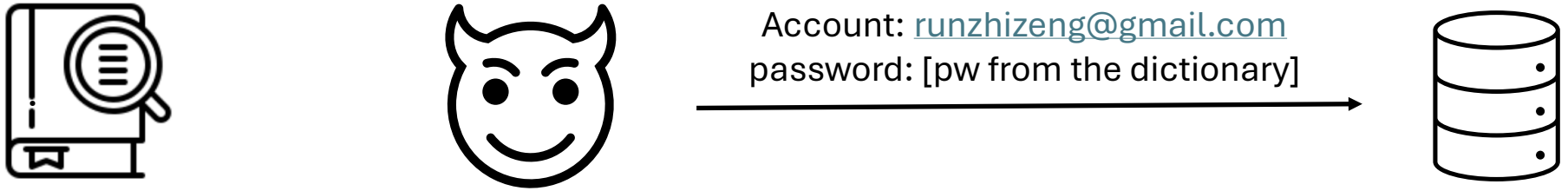
Password Dictionary

- Construct a password dictionary:



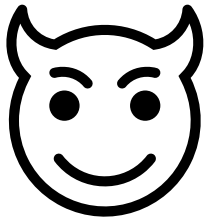
(Passwords was stored in plaintext)

Online Dictionary Attack

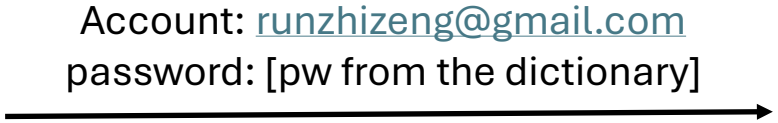


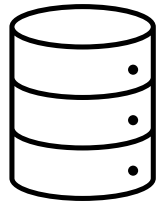
- **Online dictionary attack**
  - Attempt passwords from the dictionary until success
  - Require **Online** connections: Verify guess via interacting with the legitimate system

Online Dictionary Attack



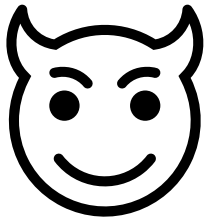
Account: runzhizeng@gmail.com
password: [pw from the dictionary]





- **Online dictionary attack**
  - Attempt passwords from the dictionary until success
  - Require **Online** connections: Verify guess via interacting with the legitimate system
  - **Unavoidable** (in most of cases), but **Detectable** and **Accountable**
  - Non-cryptographic solution: Limit failed trials

Offline Dictionary Attack

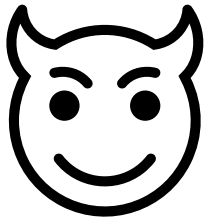


func_pw
= **F**(“RunzhiZeng123456”)

F is some publicly known function with collision resistance

- **Offline dictionary attack**

Offline Dictionary Attack



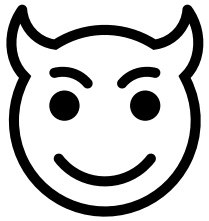
Try all passwords from the dictionary until find a pw such that **F**(pw) = func_pw

func_pw
= **F**(“RunzhiZeng123456”)

F is some publicly known function with collision resistance

- **Offline dictionary attack**
  - Attempt passwords from the dictionary until success

Offline Dictionary Attack



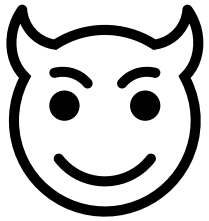
Try all passwords from the dictionary until find a pw such that **F**(pw) = func_pw

func_pw
= **F**(“RunzhiZeng123456”)

F is some publicly known function with collision resistance

- **Offline dictionary attack**
  - Attempt passwords from the dictionary until success
  - **Offline-Performable**: Verify guess without interacting with the legitimate system
  - **Hard to detect and account**

Offline Dictionary Attack



Try all passwords from the dictionary until find a pw such that **F**(pw) = func_pw

func_pw
= **F**(“RunzhiZeng123456”)

F is some publicly known function with collision resistance

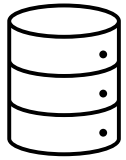
- **Offline dictionary attack**
  - Attempt passwords from the dictionary until success
  - **Offline-Performable**: Verify guess without interacting with the legitimate system
  - **Hard to detect and account**
  - **Primary Goal** of designing secure password-based cryptosystems: resist offline attacks

Offline Dictionary Attack

- **Example:** Does this login system resist offline attacks?

Account = "admin"
password = pw
where pw is some string

H is some secure hash function



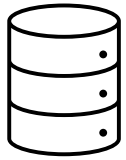
User	password
admin	$H(pw)$
Runzhi	$H(pw_1)$
Tom	$H(pw_2)$
…	…

Offline Dictionary Attack

- **Example:** Does this login system resist offline attacks?

Account = "admin"
password = pw
where pw is some string

H is some secure hash function



User	password
admin	$H(pw)$
Runzhi	$H(pw_1)$
Tom	$H(pw_2)$
…	…

1. hash_pw = **H**(pw)

LoginRequest = ("admin", hash_pw)

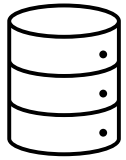
2. If hash_pw == $H(pw)$:
3. Accept
4. Else: Reject

Offline Dictionary Attack

- **Example:** Does this login system resist offline attacks?

Account = "admin"
password = pw
where pw is some string

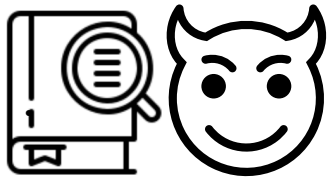
H is some secure hash function



User	password
admin	$H(pw)$
Runzhi	$H(pw_1)$
Tom	$H(pw_2)$
…	…

1. hash_pw = **H**(pw)

LoginRequest = ("admin", hash_pw)



Eavesdropping

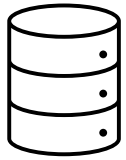
2. If hash_pw == $H(pw)$:
3. Accept
4. Else: Reject

Offline Dictionary Attack

- **Example:** Does this login system resist offline attacks?

Account = "admin"
password = pw
where pw is some string

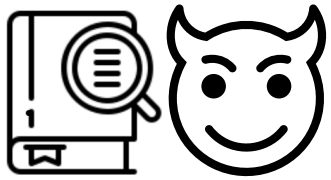
H is some secure hash function



User	password
admin	$H(pw)$
Runzhi	$H(pw_1)$
Tom	$H(pw_2)$
…	…

1. hash_pw = **H**(pw)

LoginRequest = ("admin", hash_pw)



Eavesdropping

2. If hash_pw == $H(pw)$:
3. Accept
4. Else: Reject

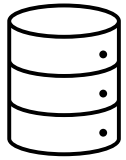
Try all pw from the dictionary until find a match: **H**(pw) == hash_pw

Offline Dictionary Attack

- **Example:** Does this login system resist offline attacks?

Account = "admin"
password = pw
where pw is some string

K is some publicly known symmetric key



User	password
admin	pw
Runzhi	pw_1
Tom	pw_2
…	…

1. enc_pw = AEAD(**K**, pw)

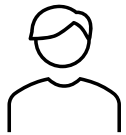
LoginRequest = ("admin", enc_pw)

2. local_enc_pw = AEAD(**K**, pw),
   // where pw is the password of "admin" from the local database
3. If local_enc_pw == enc_pw:
4. Accept
5. Else: Reject

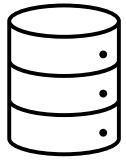
Offline Dictionary Attack

- **Example:** Does this login system resist offline attacks?

Account = "admin"
password = pw
where pw is some string



Run TLS handshake to share a handshake key K



User	password
admin	pw
Runzhi	pw_1
Tom	pw_2
…	…

1. enc_pw = AEAD(**K**, pw)

LoginRequest = ("admin", enc_pw)

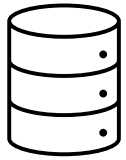
2. local_enc_pw = AEAD(**K**, pw),
   // where pw is the password of "admin" from the local database
3. If local_enc_pw == enc_pw:
4. Accept
5. Else: Reject

Offline Dictionary Attack

- **Example:** Does this login system resist offline attacks?

Account = "admin"
password = pw
where pw is some string

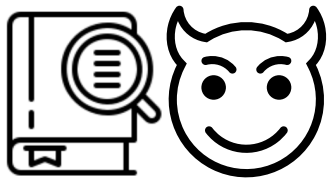
Run TLS handshake to share a handshake key K



User	password
admin	pw
Runzhi	pw_1
Tom	pw_2
…	…

1. enc_pw = AEAD(**K**, pw)

LoginRequest = ("admin", enc_pw)



Eavesdropping

2. local_enc_pw = AEAD(**K**, pw),
   // where pw is the password of "admin" from the local database
3. If local_enc_pw == enc_pw:
4. Accept
5. Else: Reject

A Summary about Online/Offline Dictionary Attack

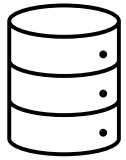
	Online Dictionary Attack	Offline Dictionary Attack
	Based on pre-constructed dictionaries	
Type of Interaction	Have to be online, one guess = one interaction with the server	Offline, can be performed locally
Accountability	Easy	Hard
Detectability	Easy	Hard
Security consideration	Unavoidable	**Primary Goal: resist offline attacks**
Solution	Restrict the number of failed attempts, …	**Need cryptographic techniques!**

Authentication using Passwords

- Most common practice: TLS + password (e.g., widely used in HTTPs login)

Account = "Runzhi"
password = pw
where pw is some string

Run TLS handshake to share a handshake key K



User	password
Runzhi	pw
Tom	pw_2
…	…

Login Request = ("Runzhi", pw)

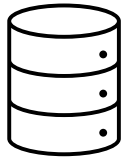
(Encrypted by the TLS handshake key K)

Authentication using Passwords

- Most common practice: TLS + password (e.g., widely used in HTTPs login)

Account = "Runzhi"
password = pw
where pw is some string

Run TLS handshake to share a handshake key K



User	password
Runzhi	pw
Tom	pw_2
…	…

Login Request = ("Runzhi", pw)

(Encrypted by the TLS handshake key K)

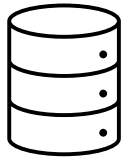
- Advantage: Easy to implement, rely on TLS, …
- Disadvantage: **Passwords are stored in plaintext**

Authentication using Passwords

- Most common practice: TLS + **hashed** password

Account = "Runzhi"
password = pw
where pw is some string

Run TLS handshake to share a handshake key K



User	password
Runzhi	$H(pw)$
Tom	$H(pw_2)$
…	…

Login Request = ("Runzhi", **H(pw)**)

(Encrypted by the TLS handshake key **K**)

- Now the server stores the hashes of passwords…
- **What happens if the database is compromised?**

Authentication using Passwords

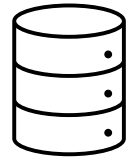
- Most common practice: TLS + hashed password

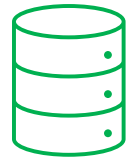
Account = "Runzhi"
password = pw
where pw is some string

Run TLS handshake to share a handshake key K

Login Request = ("Runzhi", **H(pw)**)

(Encrypted by the TLS handshake key **K**)



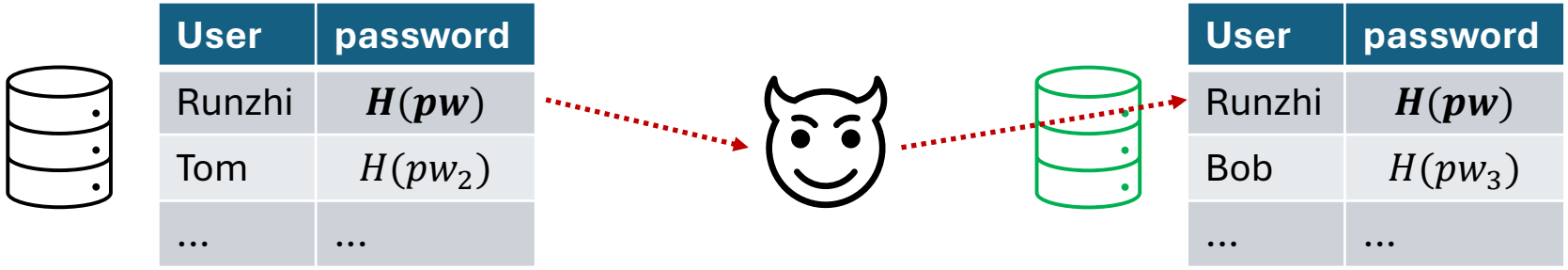


User	password
Runzhi	$H(pw)$
Tom	$H(pw_2)$
...	...

User	password
Runzhi	$H(pw)$
Bob	$H(pw_3)$
...	...

- Now the server stores the hashes of passwords…
- **Generally, passwords are reused across different servers…**

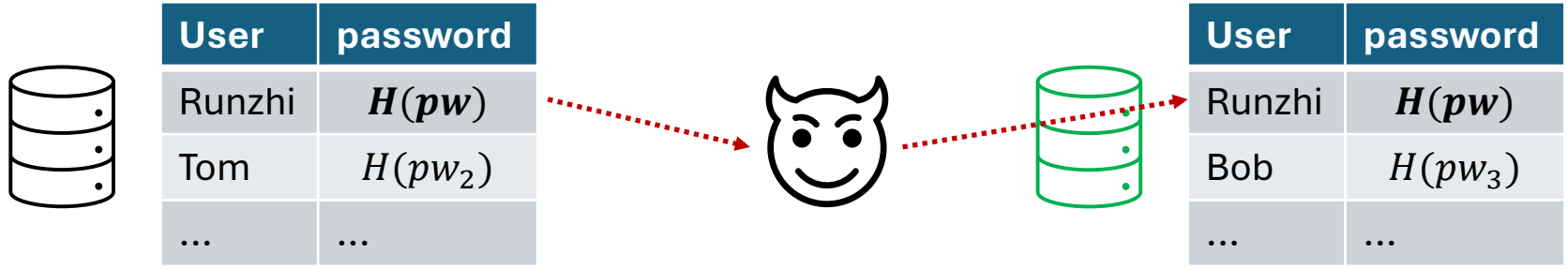
Password Storage and Salting



Store hashes of passwords v.s Store passwords in plaintext

- The former one is almost as insecure as the latter one if different servers store hashes of passwords
- **Why:** Just storing hashes can lead to cross-system compromise, making it nearly as insecure as storing plaintext passwords.

Password Storage and Salting



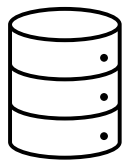
Store hashes of passwords v.s Store passwords in plaintext

- The former one is almost as insecure as the latter one if different servers store hashes of passwords
- **Why:** Just storing hashes can lead to cross-system compromise, making it nearly as insecure as storing plaintext passwords.

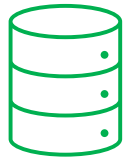
- **Solution: Salting (i.e., store salted hashes of passwords)**

Password Storage and Salting

User	password
Runzhi	$H(pw)$
Tom	$H(pw_2)$
...	...

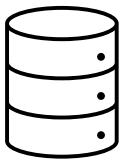


User	password
Runzhi	$H(pw)$
Bob	$H(pw_3)$
...	...

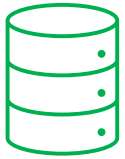


Password Storage and Salting

User	password
Runzhi	$r, H(r, pw)$
Tom	$r_2, H(r_2, pw_2)$
…	…

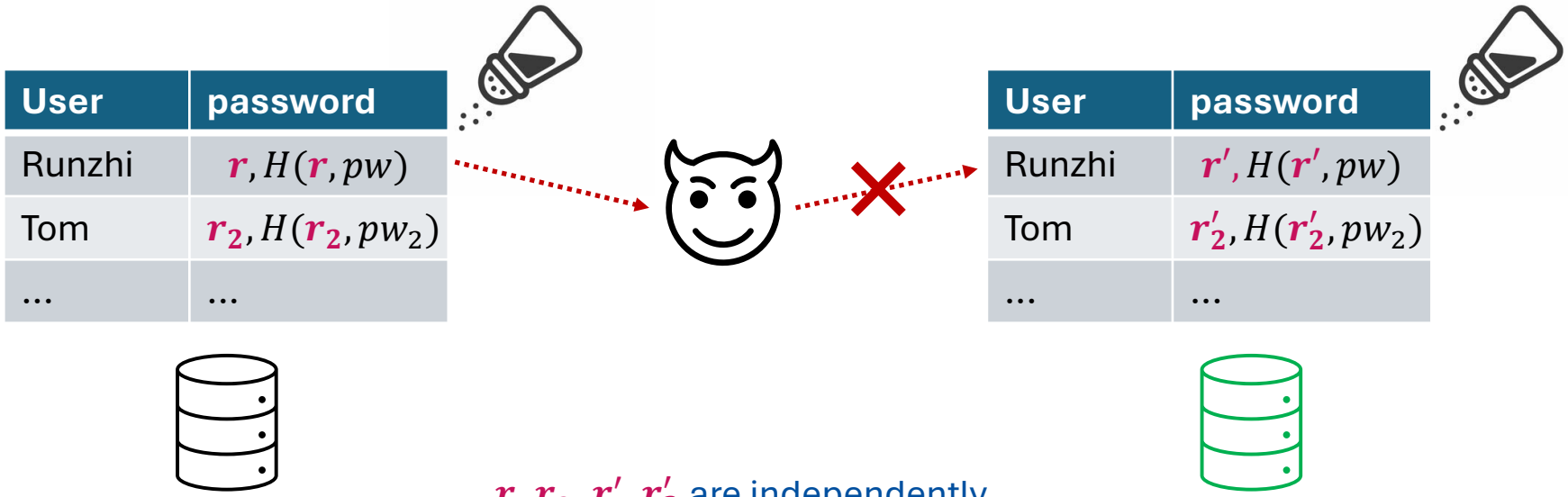


User	password
Runzhi	$r', H(r', pw)$
Tom	$r'_2, H(r'_2, pw_2)$
…	…



r, r_2, r', r'_2 are independently random strings **(salt)**

Password Storage and Salting



r, r_2, r', r_2' are independently random strings **(salt)**

- **Resistance to cross-system compromise**

Authentication using Salted Hashes of Passwords

- TLS + salted hashes password

Account = "Runzhi"
password = pw
where pw is some string

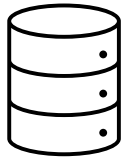
Run TLS handshake to share a handshake key K

LoginRequest = "Runzhi"

r

H(r, pw)

(Encrypted by the TLS handshake key K)



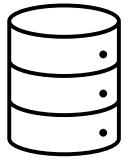
User	password
Runzhi	$r, H(r, pw)$
Tom	$r_2, H(r_2, pw_2)$
…	…

Authentication using Salted Hashes of Passwords

- TLS + salted hashes password

Account = "Runzhi"
password = pw
where pw is some string

Run TLS handshake to share a handshake key K



User	password
Runzhi	$r, H(r, pw)$
Tom	$r_2, H(r_2, pw_2)$
…	…

LoginRequest = "Runzhi"

The server should send the salt of the user in every time it logs in

r

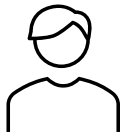
$H(r, pw)$

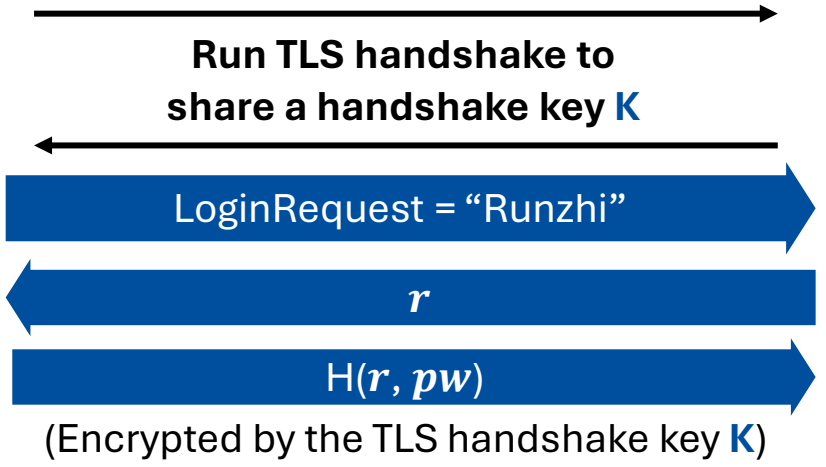
(Encrypted by the TLS handshake key K)

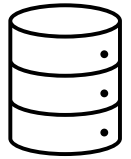
TLS + Salted Hashes of Passwords

- TLS + salted & hashed passwords
  - Use TLS to protect the transmission of pw
  - No TLS handshake key => Cannot launch offline dictionary attacks

Account = "Runzhi"
password = pw
where pw is some string







User	password_file
Runzhi	$r, H(r, pw)$
Tom	$r_2, H(r_2, pw_2)$
…	…

TLS + Salted Hashes of Passwords

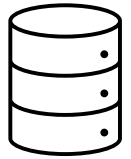
- TLS + salted & hashed passwords
  - Use TLS to protect the transmission of pw
  - No TLS handshake key => Cannot launch offline dictionary attacks

Account = "Runzhi"
password = pw
where pw is some string

Run TLS handshake to share a handshake key K

LoginRequest = "Runzhi"

r



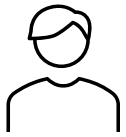
User	password_file
Runzhi	$r, H(r, pw)$
Tom	$r_2, H(r_2, pw_2)$
…	…

If the database is compromised,
then one can launch offline dictionary attack…

TLS + Salted Hashes of Passwords

- TLS + salted & hashed passwords
  - Use TLS to protect the transmission of pw
  - No TLS handshake key => Cannot launch offline dictionary attacks

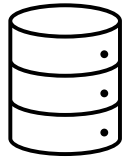
Account = "Runzhi"
password = pw
where pw is some string



Run TLS handshake to share a handshake key K

LoginRequest = "Runzhi"

r



User	password_file
Runzhi	$r, H(r, pw)$
Tom	$r_2, H(r_2, pw_2)$
…	…

If the database is compromised,
then one can launch offline dictionary attack…

Is it possible to increase the difficulty of offline attacks?

The SCRAM protocol

- Salted Challenge Response Authentication Mechanism

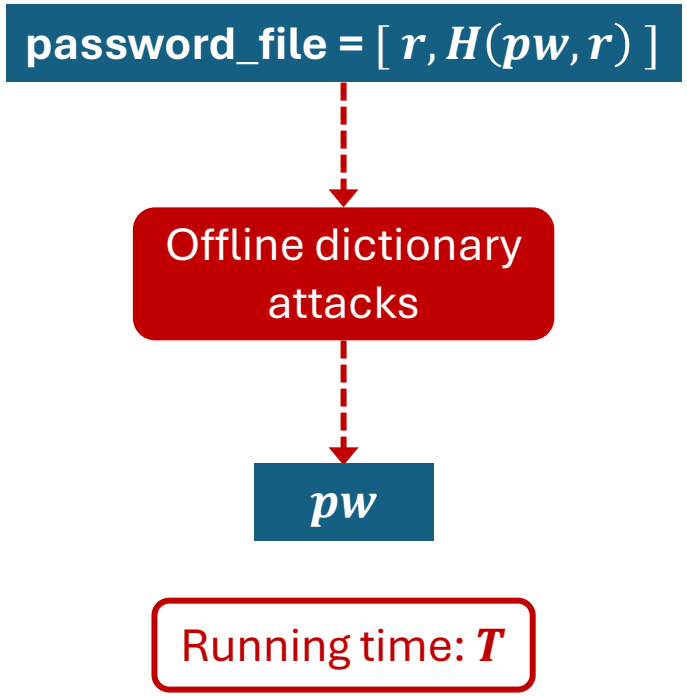
- Main idea:
  1. Add iteration in computing salted & hashed password
  2. Challenge-response Mechanism
  3. Run over TLS

- Other Important Features:
  - ➢ Inherent Resistance to Replay Attacks

    (TLS + salted & hashed passwords resists replay attacks because of TLS, while SCRAM resists replay attacks inherently, independent of the transport layer.)
  - ➢ Mutual Authentication

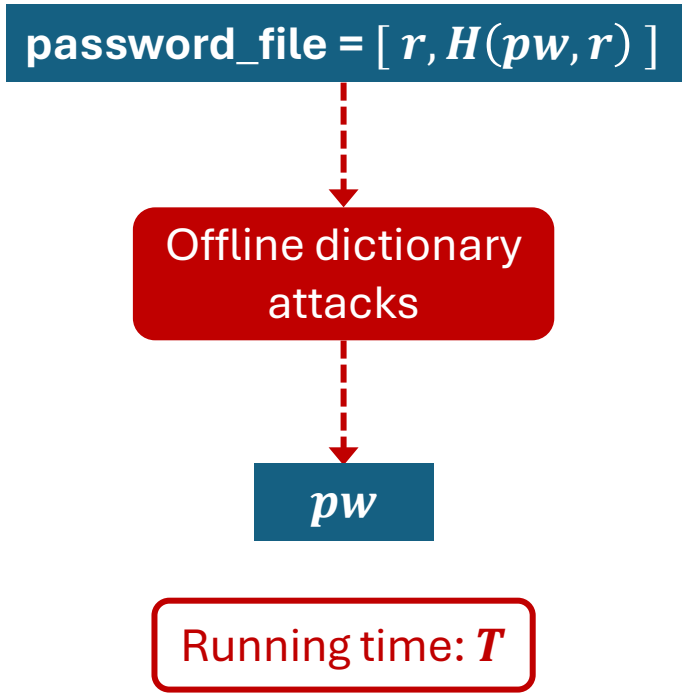
The SCRAM protocol

- Add iteration in computing salted & hashed password:



The SCRAM protocol

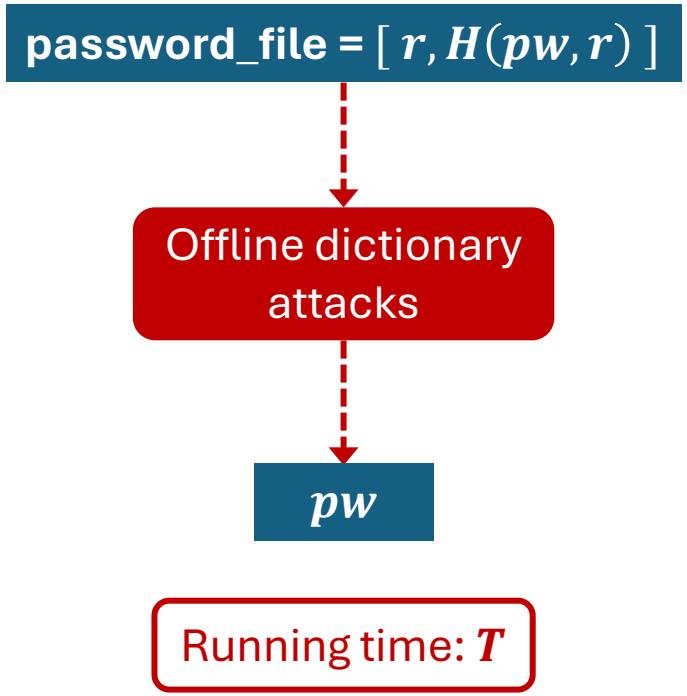
- Add iteration in computing salted & hashed password:

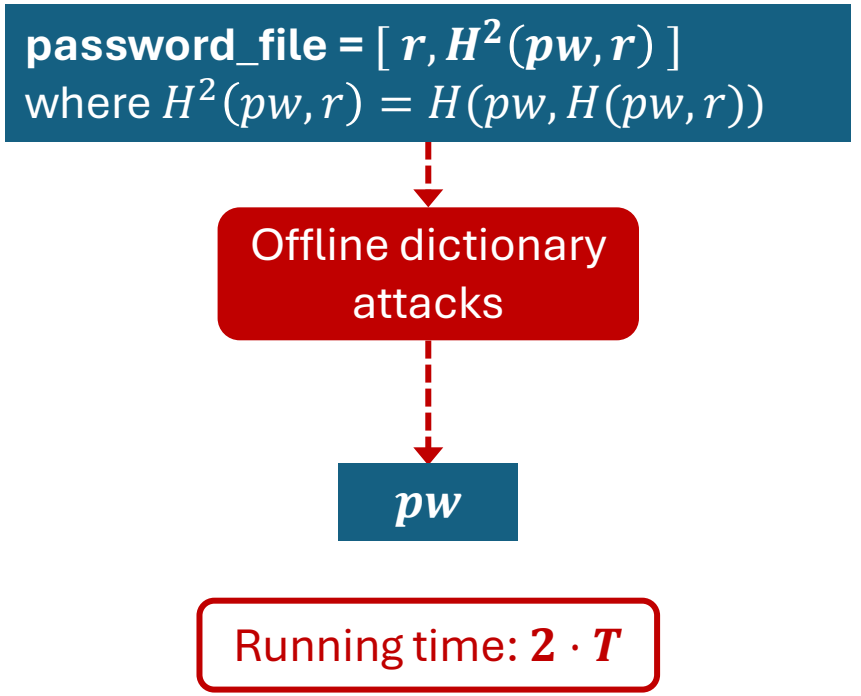


password_file = [$r, H^2(pw, r)$]
where $H^2(pw, r) = H(pw, H(pw, r))$

The SCRAM protocol

- Add iteration in computing salted & hashed password:





The SCRAM protocol

- Add iteration in computing salted & hashed password:

Iterate_hash_with_salt(*password*, *salt*, *num_of_iteration*):
// *salt* can be **16- or 32**-byte
// *num_of_iteration* can be **4096** or even **100,000**
// All variable are bytes with big-endian order

pw = *password*
padded_salt = *salt* || b'\x00\x00\x00\x01' // Append a 4-byte string 0x00000001 (in hex)

$hash_1 = \mathrm{HMAC}(pw, padded\_salt)$ // We use keyed HMAC, where the key to HMAC is the password
For i from 2 to *num_of_iteration:* // Iteratively evaluate the HMAC of pw and previous HMAC
$hash_i = \mathrm{HMAC}(pw, hash_{i-1})$

Password_file = $hash_1 \oplus hash_2 \oplus \cdots \oplus hash_{\mathrm{num\_of\_iteration}}$ // One integrate this part into the loop
return Password_file

The SCRAM protocol

- Add iteration in computing salted & hashed password:

A simpler description:
(using the notation $H^n(pw, r)$ = Iterate_hash_with_salt(pw, r, n)

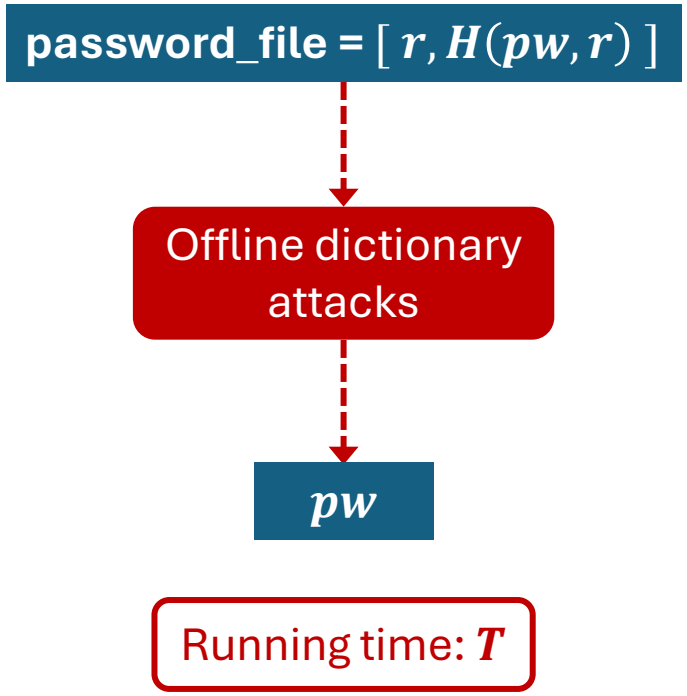
Given r, n, pw:

$$
\begin{aligned}
\mathrm{U}_1 &= \mathbf{HMAC}(pw, r \,||\, \text{b'\x00\x00\x00\x01'}) \\
\mathrm{U}_2 &= \mathbf{HMAC}(pw, \mathrm{U}_1) \\
&\vdots \\
\mathrm{U}_{n-1} &= \mathbf{HMAC}(pw, \mathrm{U}_{n-2}) \\
\mathrm{U}_n &= \mathbf{HMAC}(pw, \mathrm{U}_{n-1})
\end{aligned}
$$

We compute $H^n(pw, r) = \mathrm{U}_1 \oplus \mathrm{U}_2 \oplus \cdots \oplus \mathrm{U}_{n-1} \oplus \mathrm{U}_n$

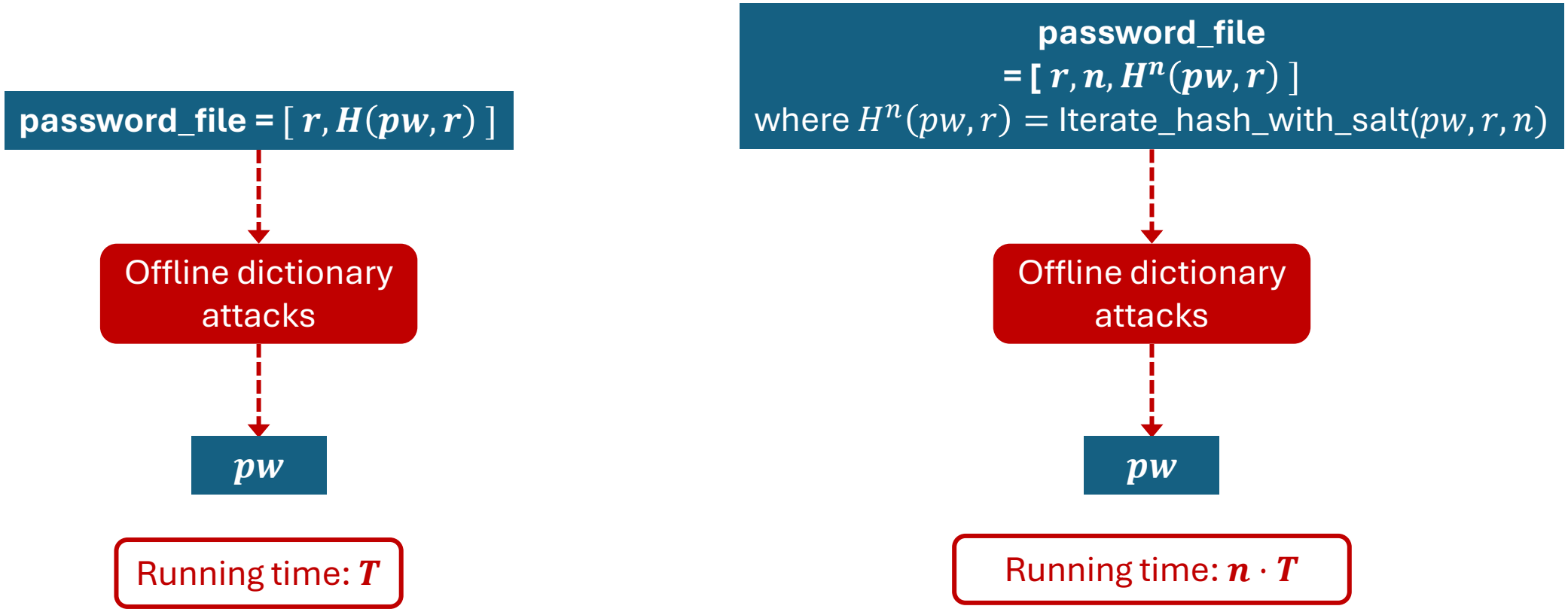
The SCRAM protocol

- Add iteration in computing salted & hashed password:



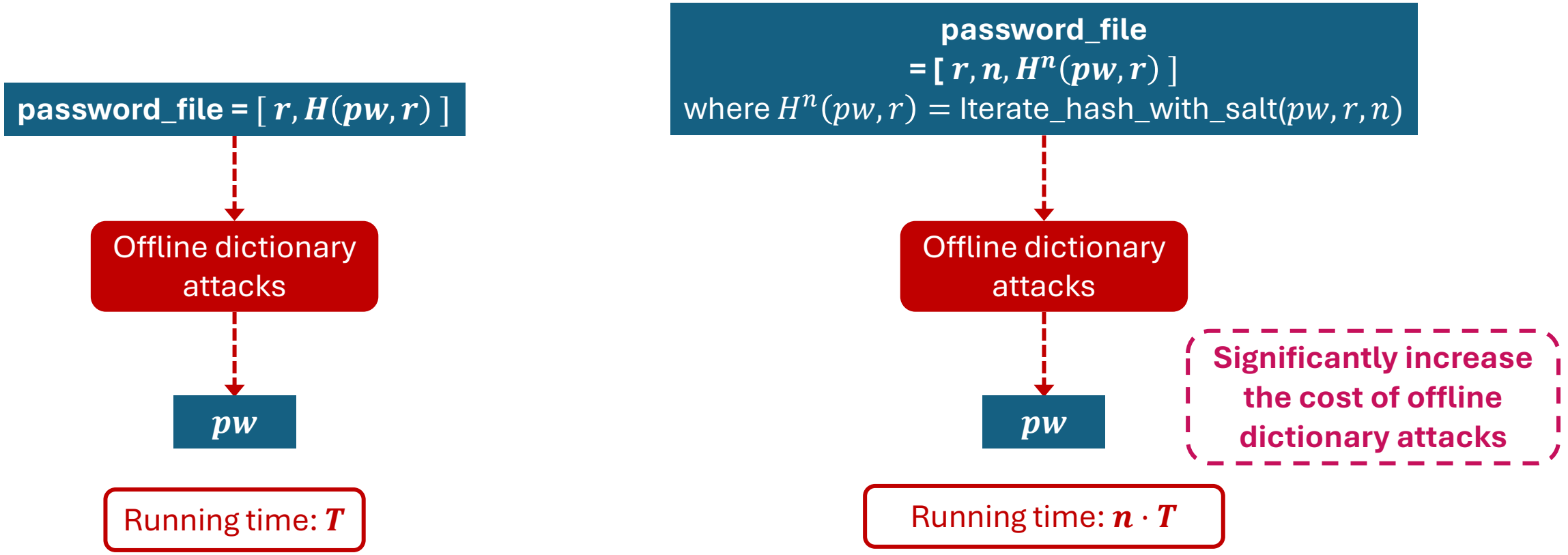
The SCRAM protocol

- Add iteration in computing salted & hashed password:



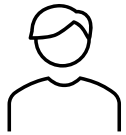
The SCRAM protocol

- Add iteration in computing salted & hashed password:

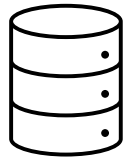


The SCRAM protocol

- Challenge-response paradigm



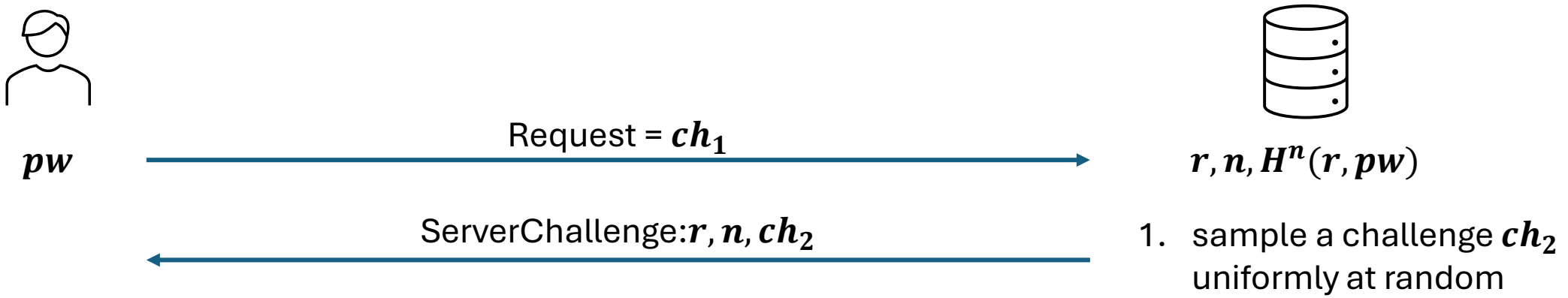
pw



$r, n, H^n(r, pw)$

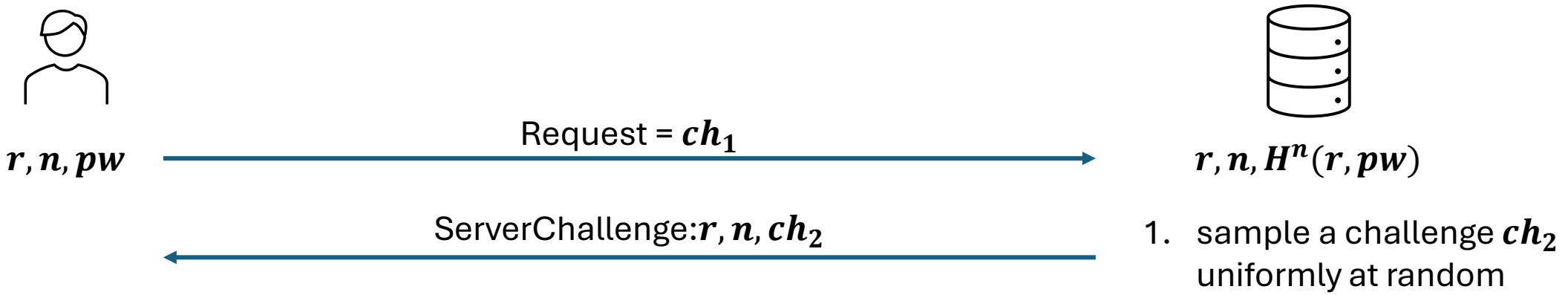
The SCRAM protocol

- Challenge-response paradigm: Client-proof



The SCRAM protocol

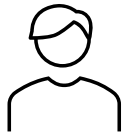
- Challenge-response paradigm: Client-proof



2. *Salted_pw* = $H^n(r, pw)$
3. *Client_key* = $\mathbf{HMAC}$(*Salted_pw*, "Client key")
4. *Auth_msg* = [Client's Name] || r, n, ch_1, ch_2
5. *Client_sign* = $\mathbf{HMAC}(\mathbf{H}(Client\_key), Auth\_msg)$ // Here $\mathbf{H}$ is the hash function used in $\mathbf{HMAC}$
6. ***Client_proof*** = *Client_key* $\oplus$ *Client_sign*

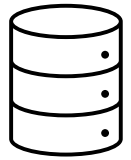
The SCRAM protocol

- Challenge-response paradigm: Client-proof



r, n, pw

Request = ch_1



$r, n, H^n(r, pw)$

ServerChallenge:r, n, ch_2

1. sample a challenge ch_2 uniformly at random

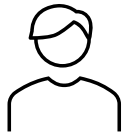
2. *Salted_pw* = $H^n(r, pw)$
3. *Client_key* = $\mathbf{HMAC}$(*Salted_pw*, "Client key")
4. *Auth_msg* = [Client's Name] || r, n, ch_1, ch_2
5. *Client_sign* = $\mathbf{HMAC}(\mathbf{H}(Client\_key), Auth\_msg)$ // Here $\mathbf{H}$ is the hash function used in $\mathbf{HMAC}$
6. ***Client_proof*** = *Client_key* $\oplus$ *Client_sign*

ClientProof: ***Client_proof***

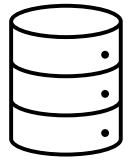
6. Verify ***Client_proof***

The SCRAM protocol

- Challenge-response paradigm: Server-sign



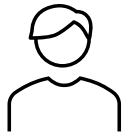
r, n, pw



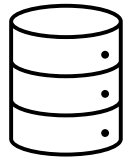
$r, n, H^n(r, pw)$

The SCRAM protocol

- Challenge-response paradigm: Server-sign



r, n, pw



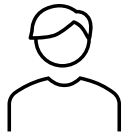
$r, n, H^n(r, pw)$

1. sample a challenge ch_1 uniformly at random

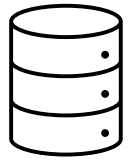
ClientChallenge: ch_1 →

The SCRAM protocol

- Challenge-response paradigm: Server-sign



r, n, pw



$r, n, H^n(r, pw)$

1. sample a challenge ch_1 uniformly at random

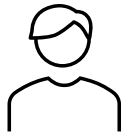
ClientChallenge: ch_1

2. *Salted_pw* = $H^n(r, pw)$
3. *Server_key* = $\mathbf{HMAC}$(*Salted_pw*, 'Client key')
4. *Auth_msg* = [Client's Name] || ch_1
5. ***Server_sign***= $\mathbf{HMAC}$(*Server_key*, *Auth_msg*)

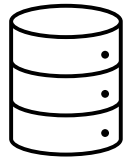
ServerSign: ***Server_sign***

The SCRAM protocol

- Challenge-response paradigm: Server-sign



$\boldsymbol{r, n, pw}$



$\boldsymbol{r, n, H^n(r, pw)}$

1. sample a challenge $\boldsymbol{ch_1}$ uniformly at random

ClientChallenge: $\boldsymbol{ch_1}$

2. *Salted_pw* = $\boldsymbol{H^n(r, pw)}$
3. *Server_key* = $\mathbf{HMAC}$(*Salted_pw*, 'Client key')
4. *Auth_msg* = [Client's Name] || $\boldsymbol{ch_1}$
5. ***Server_sign***= $\mathbf{HMAC}$(*Server_key, Auth_msg*)

ServerSign: ***Server_sign***

6. Verify ***Server_sign***

The SCRAM protocol

Account = [ClientName]
password = pw
where pw is some string

User	password_file
Runzhi	$r, n, H^n(r, pw)$
Tom	$r_2, n, H^n(r_2, pw_2)$
…	…

Run TLS handshake to share a handshake key K and some channel binding info TLS_INFO

1. Pick a random client challenge ch_1

ClientFirst: [ClientName], ch_1

2. Pick a random server challenge ch_2

ServerFirst: $ch_1||ch_2, r, n$

3. Compute ***Client_proof*** using **Auth_msg**

ClientFinal: TLS_INFO, $ch_1||ch_2$, ***Client_proof***

4. Verify ***Client_proof.***
If valid:
Compute ***Server_sign*** using **Auth_msg**

ServerFinal: ***Server_sign***

5. Verify ***Server_sign***

Auth_msg = [ClientName] || $ch_1||ch_2$ || r || n || TLS_INFO

Coding tasks

- Implement the SCRAM protocol and use your TLS implementation to protect it.