Computer Security

08. Authentication

Paul Krzyzanowski

Rutgers University

Fall 2019

Authentication

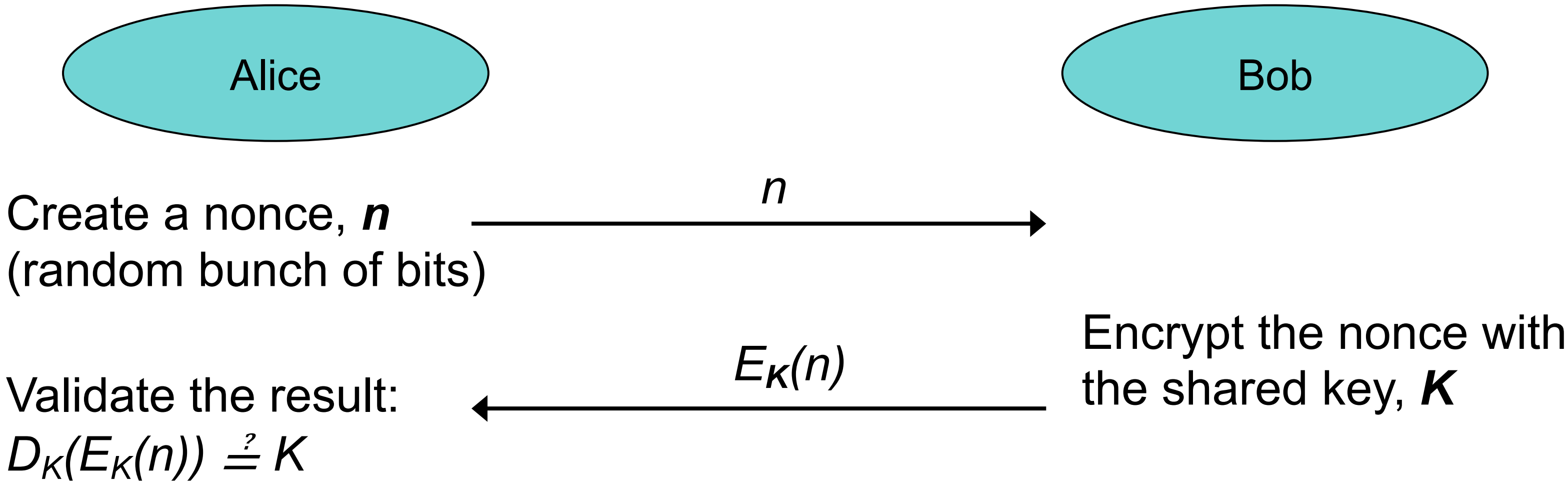
- Identification: who are you?
- Authentication: prove it
- Authorization: you can do it

Some protocols (or services) combine all three

Cryptographic Authentication

Basic concept: prove you have the key

Ask the other side to prove they can encrypt or decrypt a message with the key



This assumes a **pre-shared key** and symmetric cryptography.
After that, Alice can encrypt & send a **session key**.
Minimize the use of the pre-shared key.

Mutual authentication

- Alice had Bob prove he has the key
- Bob may want to validate Alice as well
- Bob will do the same thing
  - Have Alice prove she has the key
    - Pre-shared key: Alice encrypts the nonce with the key
    - Public key: Alice encrypts the nonce with her private key

Combined authentication & key exchange

Basic idea with symmetric cryptography:

Use a trusted third party (Trent) that has all the keys

- Alice wants to talk to Bob: she asks Trent
  - Trent generates a session key encrypted for Alice
  - Trent encrypts the same key for Bob (ticket)
- Authentication is implicit:
  - If Alice can decrypt the session key, she proved she knows her key
  - If Alice can decrypt the session key, he proved he knows his key
- Weaknesses that we need address fix:
  - Replay attacks – add nonces – Needham-Schroeder protocol
  - Replay attacks re-using a cracked old session key
    - Add timestamps: Denning-Sacco protocol, Kerberos
    - Add session IDs at each step: Otway-Rees protocol

Key exchange algorithms

Security Protocol Notation

$Z \parallel W$

- Z concatenated with W

$X \rightarrow Y : \{ Z \parallel W \} k_{A,B}$

- X sends a message to Y
- The message is the concatenation of Z & W and is encrypted by key $k_{A,B}$, which is shared by users A & B

$X \rightarrow Y : \{ Z \} k_A \parallel \{ W \} k_{A,Y}$

- X sends a message to Y
- The message is a concatenation of Z encrypted using A's key and W encrypted by a key shared by A and Y

r_1, r_2

- nonces – strings of random bits

Bootstrap problem

- How to Alice & Bob communicate securely?
- Alice cannot send a key to Bob in the clear
  - We assume an unsecure network
- We looked at two mechanisms:
  - Diffie-Hellman key exchange
  - Public key cryptography
- Let's examine the problem some more

Simple Protocol

Use a trusted third party – Trent – who has all the keys

Trent transmits a session key to Alice and Bob

{ Request session key to Bob } k_A

Alice → Trent

{ k_S } k_A || { k_S } k_B

Alice ← Trent

{ k_S } k_B

Alice → Bob

{ m } k_S

Alice ↔ Bob

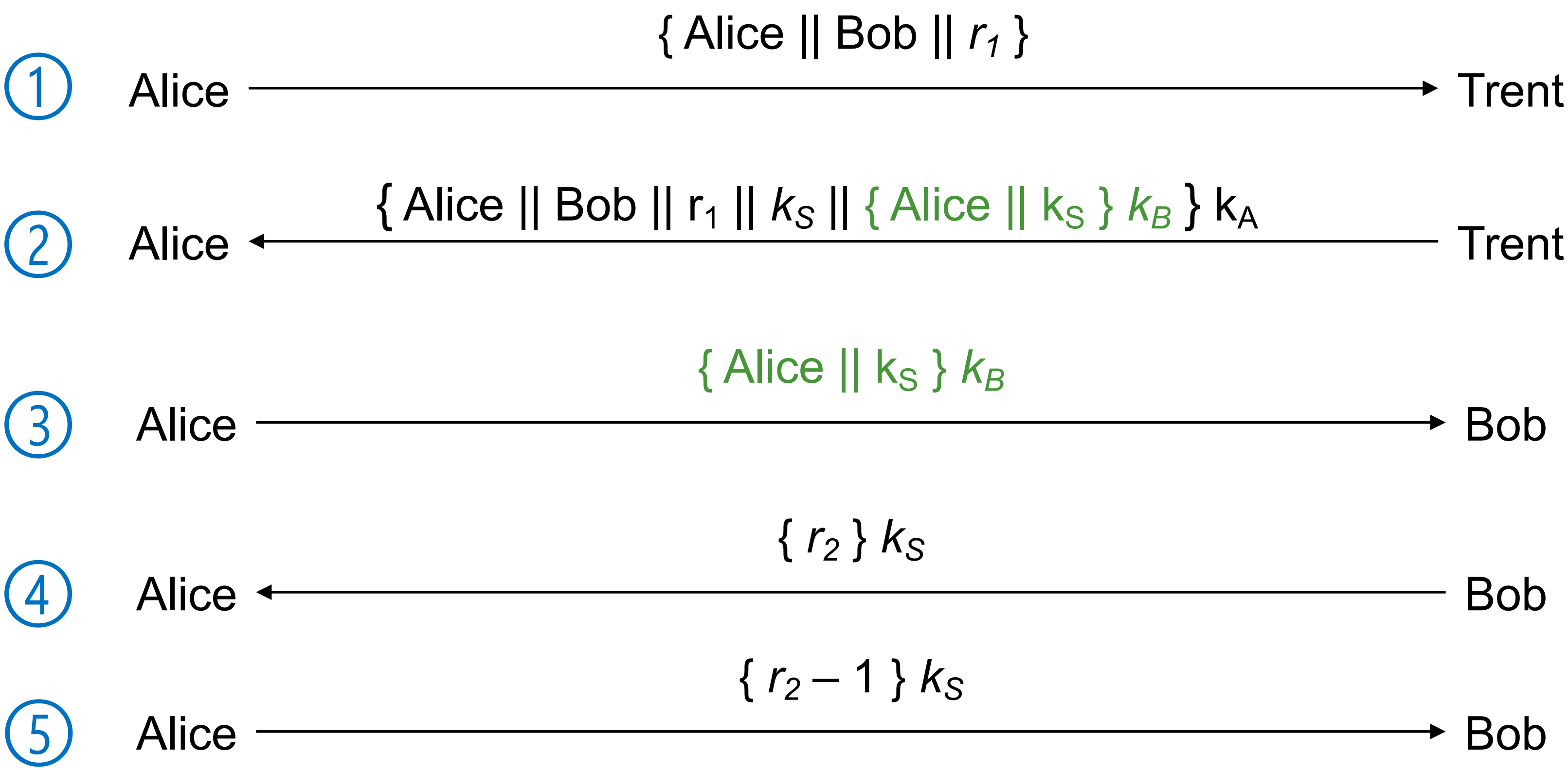
Problems

- How does Bob know he is talking to Alice?
  - Trusted third party, Trent, has all the keys
  - Trent knows the request came from Alice since only he and Alice can have the key
  - Trent can authorize Alice's request
  - Bob gets a message (session key) encrypted with his key, which only Trent could have created
    - But Bob doesn't know who requested the session
    - Trent would have to add sender information to the message

- Vulnerable to replay attacks
  - Eve records the message from Alice to Bob and later replays it
  - Bob might think he's talking to Alice, reusing the same session key
- Protocols should provide authentication & defend against replay

Needham-Schroeder

Add *nonces* – random strings – avoid replay attacks



Needham-Shroeder

Add *nonces* – random strings – avoid replay attacks

Message must have been created by Trent & is a response to the first message (contains r_1). Use of r_1 ensures it's not a replay attack.

- Alice knows only Bob & Trent can read this and get the session key.
- Bob knows it's a request from Alice

① Alice → Trent: $\{ \text{Alice} \,||\, \text{Bob} \,||\, r_1 \}$

② Alice ← Trent: $\{ \text{Alice} \,||\, \text{Bob} \,||\, r_1 \,||\, k_S \,||\, \{ \text{Alice} \,||\, k_S \} k_B \} k_A$

- Bob now tries to find out if this is a replay attack
- If it is, Eve cannot decipher r_2

③ Alice → Bob: $\{ \text{Alice} \,||\, k_S \} k_B$

This is an **authentication** step: Bob asks Alice to prove she has k_S

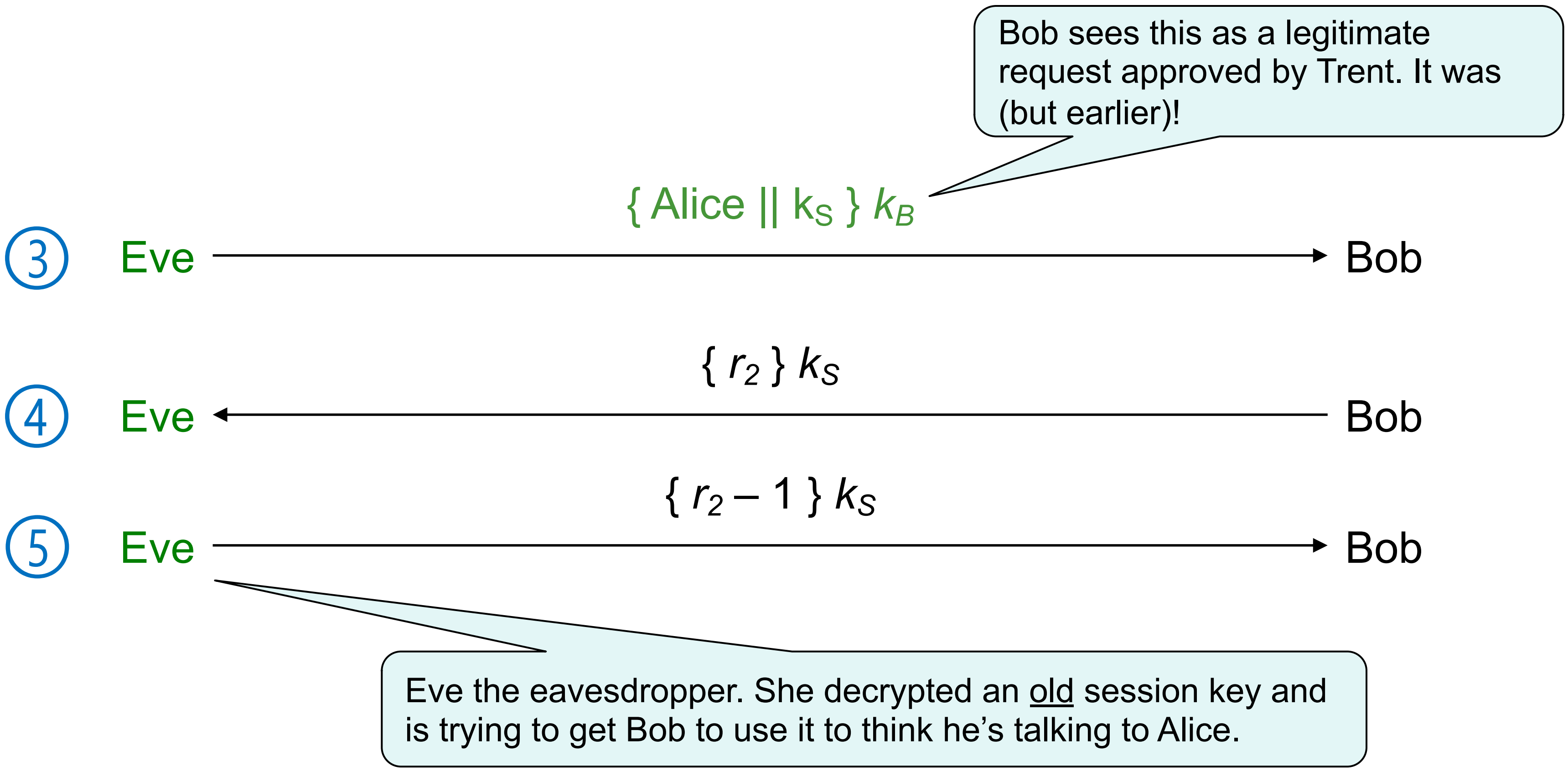
④ Alice ← Bob: $\{ r_2 \} k_S$

⑤ Alice → Bob: $\{ r_2 – 1 \} k_S$

Needham-Schroeder Protocol Vulnerability

Needham-Schroeder is still vulnerable to a certain replay attack … if an old session key is known!

- We assume all keys are secret
- But suppose Eve can obtain the session key from an <u>old</u> message (she worked hard, got lucky, and cracked an earlier message)



Denning-Sacco Solution

- Problem: replay in the third step of the protocol
  - Eve replays the message: $\{ Alice \,||\, k_S \}\, k_B$
- Solution: use a time stamp *T* to detect replay attacks
  - The trusted third party (Trent) places a timestamp in a message that is encrypted for Bob
  - The attacker has an old session key but not Alice's, Bob's or Trent's keys
  - Cannot spoof a valid message that is encrypted for Bob.

Needham-Shroeder w/Denning-Sacco mods

Add nonces – random strings – AND a timestamp

{ Alice || Bob || r_1 }

Alice → Trent

{ Alice || Bob || r_1 || k_S || { Alice || T || k_S } k_B } k_A

Alice ← Trent

{ Alice || T || k_S } k_B

Alice → Bob

{ r_2 } k_S

Alice ← Bob

{ $r_2 - 1$ } k_S

Alice → Bob

Problem with timestamps

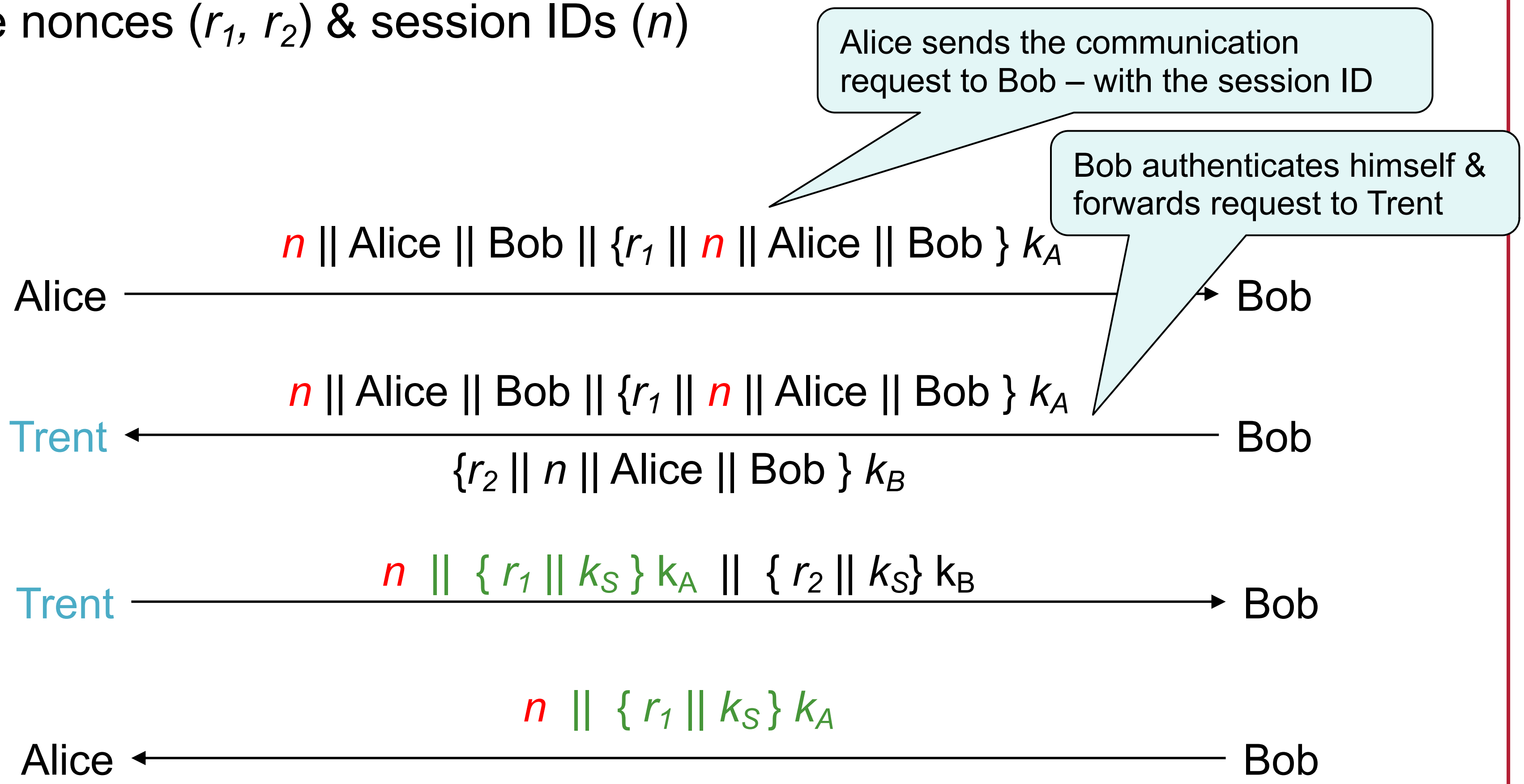
- Use of timestamps relies on synchronized clocks
  - Messages may be falsely accepted or falsely rejected because of bad time
- Time synchronization becomes an attack vector
  - Create fake NTP responses
  - Generate fake GPS signals

Otway-Rees Protocol: Session IDs

- Another way to correct the *third message replay* problem
- Instead of using timestamps
  - Use a random integer, *n*, that is associated with all messages in the key exchange
- The protocol is altered slightly
  - Alice first sends a message to Bob
    - The message contains the session ID & nonce encrypted with Alice's secret key
  - Bob forwards the message to Trent
    - And creates a message containing a nonce & the same session ID encrypted with Bob's secret key
  - Trent creates a session key & encrypts it for both Alice and for Bob

Otway-Rees Protocol

Use nonces (r_1, r_2) & session IDs (n)



Kerberos

Kerberos

- Authentication service developed by MIT
  - project Athena 1983-1988
- Uses a trusted third party & symmetric cryptography
- Based on Needham Schroeder with the Denning Sacco modification
- Passwords not sent in clear text
  - assumes only the network can be compromised

Kerberos

Users and services authenticate themselves to each other

To access a service:

- user presents a ticket issued by the Kerberos authentication server
- service examines the ticket to verify the identity of the user

Kerberos is a trusted third party

- Knows all (users and services) passwords
- Responsible for
  - Authentication: validating an identity
  - Authorization: deciding whether someone can access a service
  - Key exchange: giving both parties an encryption key (securely)

Kerberos

- User *Alice* wants to communicate with a service *Bob*
- Both Alice and Bob have keys

- Step 1:
  - Alice authenticates with Kerberos server
    - Gets *session key* and *ticket* (*sealed envelope*)
- Step 2:
  - Alice gives Bob the ticket, which contains the session key
  - Convinces Bob that she got the session key from Kerberos

Authenticate, get permission

Alice

"I'm Alice and want to talk to Bob"

{ "Alice" || Bob } →

Authentication Server (AS)

If Alice is allowed to talk to Bob,

generate session key, S

← { "Bob's server", T, k_S } k_A

Alice decrypts this:

- Gets ID of "Bob's server"
- Gets session key & timestamp
- *Knows message came from AS*

TICKET

sealed envelope

← {"Alice", T, k_S } k_B

eh? (Alice can't read this!)

Send key

Alice

Alice encrypts a timestamp with session key

$\{ \text{"Alice"}, S \}\, k_B \,||\, \{ T' \}\, k_S$

sealed envelope

Bob

Bob decrypts envelope:

- Envelope was created by Kerberos on request from Alice
- Gets session key

Decrypts time stamp

- Validates time window
- Prevent replay attacks

Authenticate recipient of message

Alice | **Bob**

Encrypt Alice's timestamp in return message

Bob → Alice: $\{ T'+1 \} k_S$

Alice validates timestamp

Alice ↔ Bob: $\{Messages\} k_S$

Alice & Bob communicate by encrypting data with S

Kerberos key usage

- Every time a user wants to access a service
  - User's password (key) must be used to decode the message from Kerberos
- We can avoid this by caching the password in a file
  - Not a good idea
- Another way: create a temporary password
  - We can cache this temporary password
  - Similar to a session key for Kerberos – to get access to other services
  - Split Kerberos server into

    Authentication Server + Ticket Granting Server

Ticket Granting Server (TGS)

- TGS works like a temporary ID
- User first requests access to the TGS
  - Contact Kerberos Authentication Server
    - Knows all users & their secret keys
    - User enters a password to do this
    - Gets back a ticket & session key to the TGS – these can be cached
- To access any service
  - Send a request to the TGS – encrypted with the TGS session key along with the ticket for the TGS
  - The ticket tells the TGS what your session key is
  - It responds with a session key & ticket for that service

Using Kerberos

`$ kinit`

`Password:` *enter password*

ask AS for permission (session key) to access TGS

Alice gets:

{"TGS", S } k_A	← *Session key*
{"Alice", S } k_{TGS}	← *TGS Ticket*
{ T } k_S	← *Encrypted timestamp*

Compute key (A) from password to decrypt session key S and get TGS ID.

You now have a ticket to access the Ticket Granting Service

Using Kerberos

`$ rlogin somehost`

rlogin uses the TGS Ticket to request a ticket for the *rlogin* service on *somehost*

Alice sends session key, S, to TGS

rlogin → $\{$"Alice", $k_S\}$ k_{TGS}, $\{T\}$ k_S → TGS

Alice receives session key for rlogin service & ticket to pass to rlogin service

← $\{$"rlogin@somehost", $k_{S'}\}$ k_S

← $\{$"Alice", $k_{S'}\}$ k_R

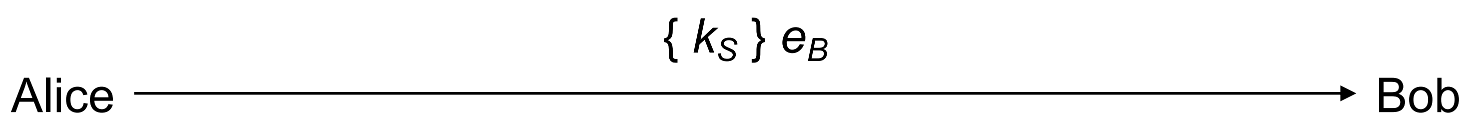
S' = session key for *rlogin*

ticket for rlogin server on somehost

Public Key Exchange

We did this

- Alice's & Bob's public keys known to all: e_A, e_B
- Alice & Bob's private keys are known only to the owner: d_a, d_b
- Simple protocol to send symmetric session key: k_S



Problem

- Vulnerable to forgery or replay
- Public keys are known to anyone
  - Bob has no assurance that Alice sent the message
- Fix: have Alice sign the session key

$\{ \{ k_S \} d_a \} e_B$

Alice ⟶ Bob

Key k_S encrypted with Alice’s private key
Entire message encrypted with Alice’s public key

Problem #2

- How do we know we have the right public keys?
- Send a certificate so Bob can verify it

$\{ \{ k_S \} d_a \} e_B$, X

Alice → Bob

Add Alice's certificate, which contains Alice's verifiable public key

Combined authentication & key exchange

- Basic idea with symmetric cryptography:

  Use a trusted third party (Trent) that has all the keys

  - Alice wants to talk to Bob: she asks Trent
    - Trent generates a session key encrypted for Alice
    - Trent encrypts the same key for Bob (ticket)
  - Authentication is implicit:
    - If Alice can decrypt the session key, she proved she knows her key
    - If Alice can decrypt the session key, he proved he knows his key
  - Weaknesses that we had to fix:
    - Replay attacks – add nonces – Needham-Schroeder protocol
    - Replay attacks re-using a cracked old session key
      - Add timestamps (Denning-Sacco protocol, Kerberos)
      - Add session IDs at each step (Otway-Rees Protocol)

Cryptographic toolbox

- Symmetric encryption
- Public key encryption
- Hash functions
- Random number generators

User Authentication

Authentication

<u>Three factors</u>:

- **Ownership**: something you have
  - *Key, card*
  - Can be stolen
- **Knowledge**: something you know
  - *Passwords, PINs*
  - Can be guessed, shared, stolen
- **Inherence**: something you are
  - *Biometrics*
  - Usually needs hardware, can be copied (sometimes)
  - Once copied, you're stuck

Multi-Factor Authentication

Factors may be combined

- ATM machine: 2-factor authentication (2FA)
  - ATM card something you have
  - PIN something you know

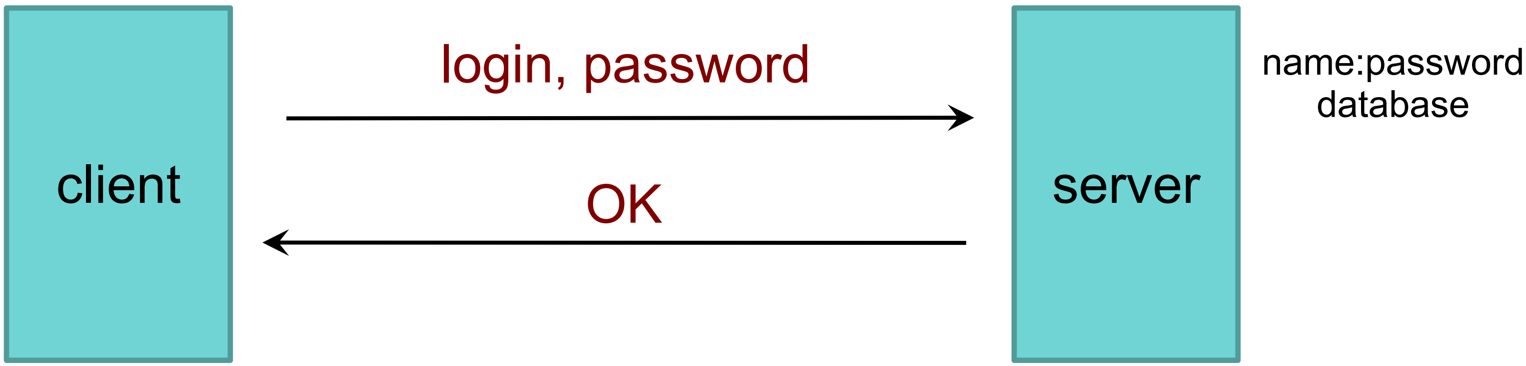
- Password + code delivered via SMS: 2-factor authentication
  - Password something you know
  - Code validates that you possess your phone

Two passwords ≠ Two-factor authentication

The factors must be different

Authentication: PAP

Password Authentication Protocol



- Unencrypted, reusable passwords
- Insecure on an open network
- Also, the password file must be protected from open access
  - But administrators can still see everyone's passwords
    What if you use the same password on Facebook as on Amazon?

Passwords are bad

- Human readable & easy to guess
  - People usually pick really bad passwords
- Easy to forget
- Usually short
- Static ... reused over & over
  - Security is as strong as the weakest link
  - If a user name (or email) & password is stolen from one server, it might be usable on others
- Replayable
  - If someone can grab it or see it, they can play it back

Recent large-scale leaks of password from servers have shown that people DO NOT pick good passwords

Common Passwords

Adobe security breach (November 2013)

- 152 million Adobe customer records … with encrypted passwords
- Adobe encrypted passwords with a symmetric key algorithm
- … and used the same key to encrypt every password!

Top 26 Adobe Passwords

	Frequency	Password
1	1,911,938	123456
2	446,162	123456789
3	345,834	password
4	211,659	adobe123
5	201,580	12345678
6	130,832	qwerty
7	124,253	1234567
8	113,884	111111
9	83,411	photoshop
10	82,694	123123
11	76,910	1234567890
12	76,186	000000
13	70,791	abc123

	Frequency	Password
14	61,453	1234
15	56,744	adobe1
16	54,651	macromedia
17	48,850	azerty
18	47,142	iloveyou
19	44,281	aaaaaa
20	43,670	654321
21	43,497	12345
22	37,407	666666
23	35,325	sunshine
24	34,963	123321
25	33,452	letmein
26	32,549	monkey

It's not getting better

Leaks have not convinced people to use good passwords

Rank	2012	2013	2014	2015	2016	2017	2018
1	password	123456	123456	123456	123456	123456	123456
2	123456	password	password	password	password	password	password
3	12345678	12345678	12345	12345678	12345	12345678	123456789
4	abc123	qwerty	12345678	qwerty	12345678	qwerty	12345678
5	qwerty	abc123	qwerty	12345	football	12345	12345
6	monkey	123456789	123456789	123456789	qwerty	123456789	111111
7	letmein	111111	1234	football	1234567890	letmein	1234567
8	dragon	1234567	baseball	1234	1234567	1234567	sunshine

Past seven years of top passwords from SplashData's list

https://en.wikipedia.org/wiki/List_of_the_most_common_passwords

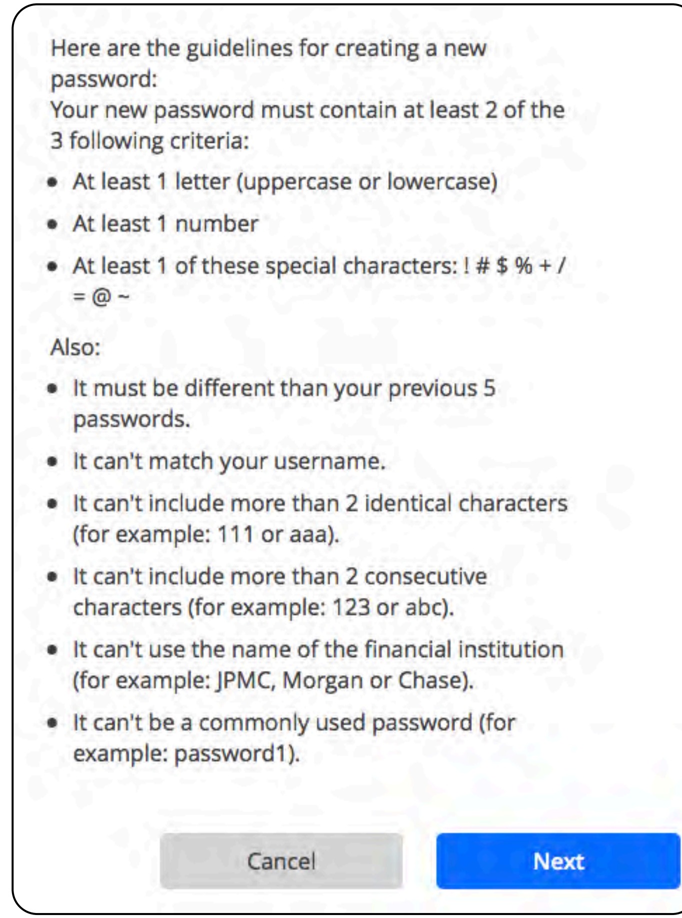
Policies to the rescue?

- Password rules

  "Everyone knows that an exclamation point is a 1, or an I, or the last character of a password. $ is an S or a 5. If we use these well-known tricks, we aren't fooling any adversary. We are simply fooling the database that stores passwords into thinking the user did something good"

  — Paul Grassi, NIST

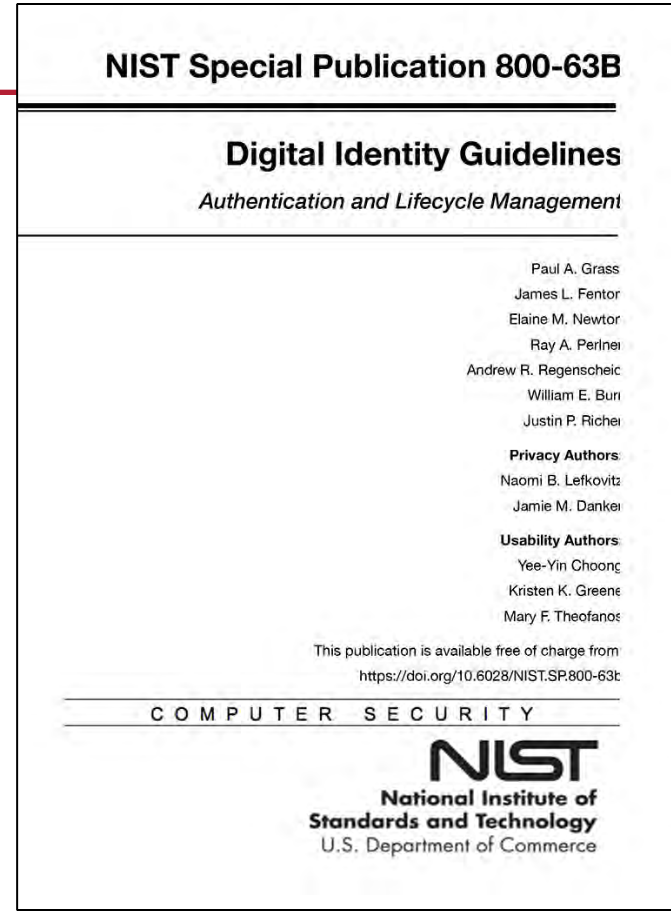
- Periodic password change requirements
  - People tend to change passwords rapidly to exhaust the history list and get back to their favorite password
  - Forbidding changes for several days enables a denial of service attack
  - People pick worse passwords, incorporating numbers, months, or years



https://fortune.com/2017/05/11/password-rules/
https://pages.nist.gov/800-63-3/sp800-63b.html#sec5

NIST recommendations

- Remove periodic password change requirements
- Drop complexity requirements (numbers, letters, symbols)
- Choose long passwords
- Avoid
  - Passwords obtained from databases of previous breaches
  - Dictionary words
  - Repetitive or sequential characters (e.g. ‘aaaaa’, ‘1234abcd’)
  - Context-specific words, such as the name of the service, the username, and derivatives thereof


NIST Special Publication 800-63B

Digital Identity Guidelines

Authentication and Lifecycle Management

Paul A. Grass
James L. Fentor
Elaine M. Newtor
Ray A. Perlner
Andrew R. Regenscheic
William E. Burr
Justin P. Richer

Privacy Authors
Naomi B. Lefkovitz
Jamie M. Danker

Usability Authors
Yee-Yin Choong
Kristen K. Greene
Mary F. Theofanos

This publication is available free of charge from
https://doi.org/10.6028/NIST.SP.800-63b

COMPUTER SECURITY

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

https://pages.nist.gov/800-63-3/sp800-63b.html

PAP: Reusable passwords

Problem #1: Open access to the password file

What if the password file isn't sufficiently protected and an intruder gets hold of it? All passwords are now compromised!

Even if a trusted admin sees your password, this might also be your password on other systems.

How about encrypting the passwords?

- Where would you store the key?
- Adobe did that
  - 2013 Adobe security breach leaked 152 million Adobe customer records
  - Adobe used encrypted passwords
    - But the **passwords were all encrypted with the same key**
    - If the attackers steal the key, they get the passwords

PAP: Reusable passwords

Solution:

Store a hash of the password in a file

- Given a file, you don't get the passwords
- Have to resort to a dictionary or brute-force attack
- Example, passwords hashed with SHA-512 hashes (SHA-2)

What is a dictionary attack?

- **Suppose you got access to a list of hashed passwords**
- **Brute-force, exhaustive search: try every combination**
  - Letters (A-Z, a-z), numbers (0-9), symbols (!@#$%...)
  - Assume 30 symbols + 52 letters + 10 digits = 92 characters
  - Test all passwords up to length 8
  - Combinations = $92^8 + 92^7 + 92^6 + 92^5 + 92^4 + 92^3 + 92^2 + 92^1 = 5.189 \times 10^{15}$
  - If we test 1 billion passwords per second: ≈ 60 days
- **But some passwords are more likely than others**
  - 1,991,938 Adobe customers used a password = "123456"
  - 345,834 users used a password = "password"
- **Dictionary attack**
  - Test lists of common passwords, dictionary words, names
  - Add common substitutions, prefixes, and suffixes

Easiest to do if the attacker steals a hashed password file –
so we read-protect the hashed passwords to make it harder to get them

How to speed up a dictionary attack

Create a table of precomputed hashes

Now we just search a table for the hash to find the password

SHA-256 Hash	**password**
8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92	123456
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8	password
ef797c8118f02dfb649607dd5d3f8c7623048c9c063d532cc95c5ed7a898a64f	12345678
1c8bfe8f801d79745c4631d09fff36c82aa37fc4cce4fc946683d7b336b63032	letmein
…	…

Salt: defeating dictionary attacks

Salt = random string (typically up to 16 characters)

- Concatenated with the password
- Stored with the password file (it's not secret)

"**am$7b22QL**" + "**password**"

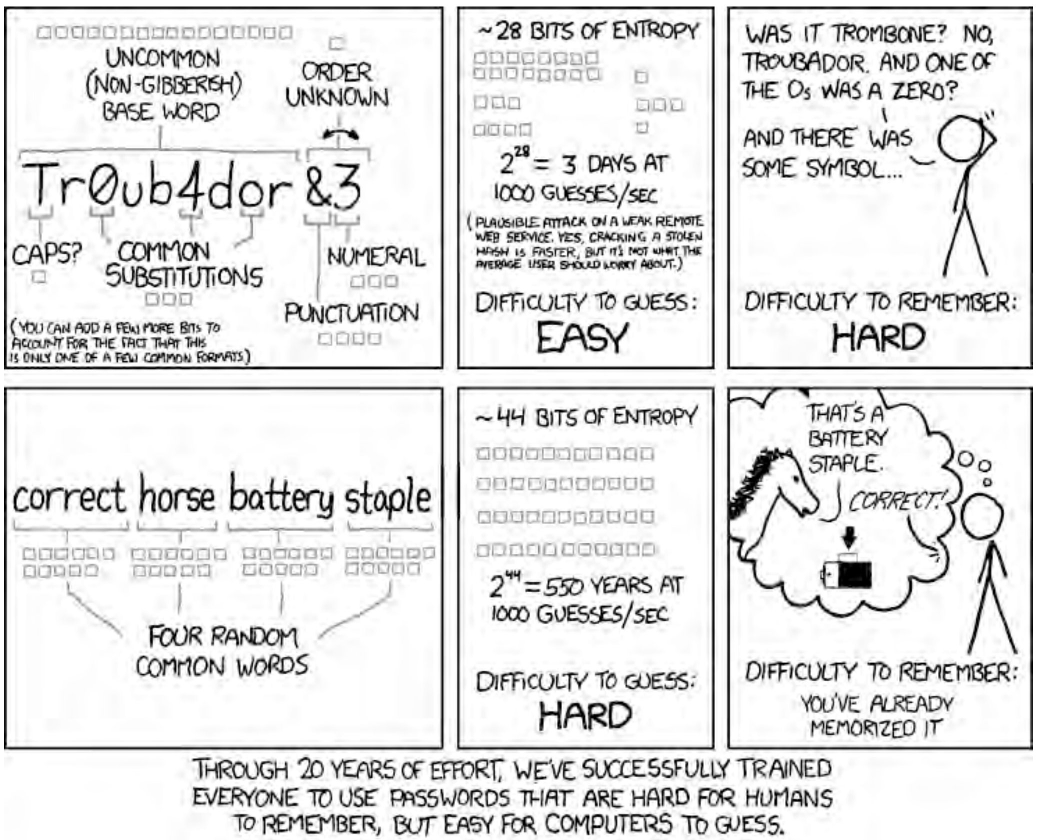
- Even if you know the salt, you cannot use precomputed hashes to search for a password
  (because the salt is prefixed to the password string)

Example: SHA-256 hash of "**password**", salt = "**am$7b22QL**"=
hash("**am$7b22QLpassword**")=
7a87d1d5118873b1c16d30176936e1920f33b91d8be1517d5cc295dfd0268906

*You will **not** have a precomputed hash("am$7b22QLpassword")*

Longer passwords

- English text has an entropy of about 1.2-1.5 bits per character
- Random text has an entropy ≈ $\log_2(1/95)$ ≈ 6.6 bits/character



Assume 95 printable characters

Defenses

- Use longer passwords
  - But can you trust users to pick ones with enough entropy?
- Rate-limit guesses
  - Add timeouts after an incorrect password
    - Linux waits about 3 secs – and terminates the *login* program after 5 tries
- Lock out the account after *N* bad guesses
  - But this makes you vulnerable to denial-of-service attacks
- Use a slow algorithm to make guessing slow

People forget passwords

- Especially seldom-used ones
- How do we handle that?
- Email them?
  - Common solution
  - Requires that the server be able to get the password (can't store a hash)
  - What if someone reads your email?
- Reset them?
  - How do you authenticate the requester?
  - Usually send reset link to email address created at registration
  - But – what if someone reads your mail? …or you no longer have that address?
- Provide hints?
- Write them down?
  - OK if the threat model is electronic only

Reusable passwords in multiple places

- People often use the same password in different places
- If one site is compromised, the password can be used elsewhere
  - People often try to use the same email address and/or user name
- This is the root of phishing attacks
- Password managers
  - Software that stores passwords in an encrypted file
  - Do you trust the protection? The synchronization capabilities?
  - Can malware get to the database?
  - In general, these are good
    - Way better than storing passwords in a file
    - Encourages having unique passwords per site
    - Password managers may have the ability to recognize web sites & defend against phishing

9 Popular Password Manager Apps Found Leaking Your Secrets

Tuesday, February 28, 2017 Wang Wei

PAP: Reusable passwords

Problem #2: Network sniffing or shoulder surfing

Passwords can be stolen by observing a user's session in person or over a network:

- Snoop on telnet, ftp, rlogin, rsh sessions
- Trojan horse
- Social engineering
- Key logger, camera, physical proximity
- Brute-force or dictionary attacks

Solutions:

(1) Use an encrypted communication channel

(2) Use one-time passwords

(3) Use multi-factor authentication, so a password alone is not sufficient

One-time passwords

Use a different password each time

– If an intruder captures the transaction, it won't work next time

Three forms

1. Sequence-based: password = *f*(previous password)
2. Time-based: password = *f*(time, secret)
3. Challenge-based: *f*(challenge, secret)

S/key authentication

- One-time password scheme
- Produces a limited number of authentication sessions
- Relies on one-way functions

S/key authentication

<u>Authenticate Alice for 100 logins</u>

- pick random number, R
- using a one-way function, $f(x)$:

$$x_1 = f(R)$$
$$x_2 = f(x_1) = f(f(R))$$
$$x_3 = f(x_2) = f(f(f(R)))$$
$$\dots \quad \dots$$
$$x_{100} = f(x_{99}) = f(\dots f(f(f(R)))\dots)$$

Give this list to Alice

- then compute:

$$x_{101} = f(x_{100}) = f(\dots f(f(f(R)))\dots)$$

S/key authentication

<u>Authenticate Alice for 100 logins</u>

Store $\mathbf{x_{101}}$ in a password file or database record associated with Alice

alice: x_{101}

S/key authentication

Alice presents the *last* number on her list:

Alice to host: { "alice", x_{100} }

Host computes $f(x_{100})$ and compares it with the value in the database

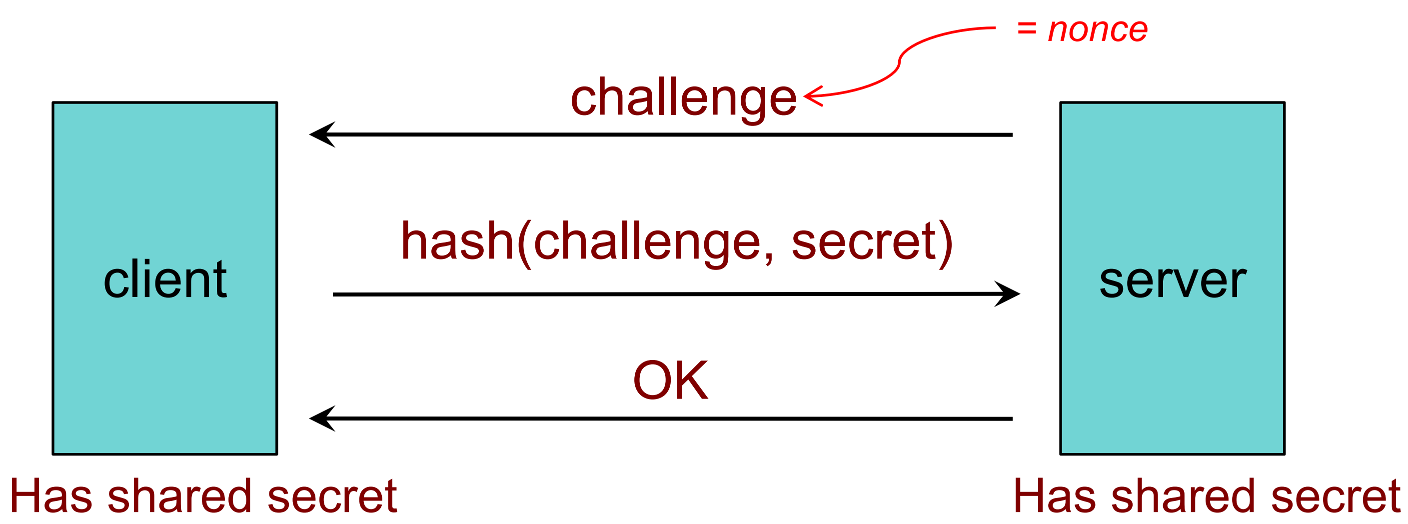
```
if (x100 provided by alice) = passwd("alice")
    replace x101 in db with x100 provided by alice
    return success
else
    fail
```

next time: Alice presents x_{99}

If someone sees x_{100} there is no way to generate x_{99}.

Authentication: CHAP

Challenge-Handshake Authentication Protocol

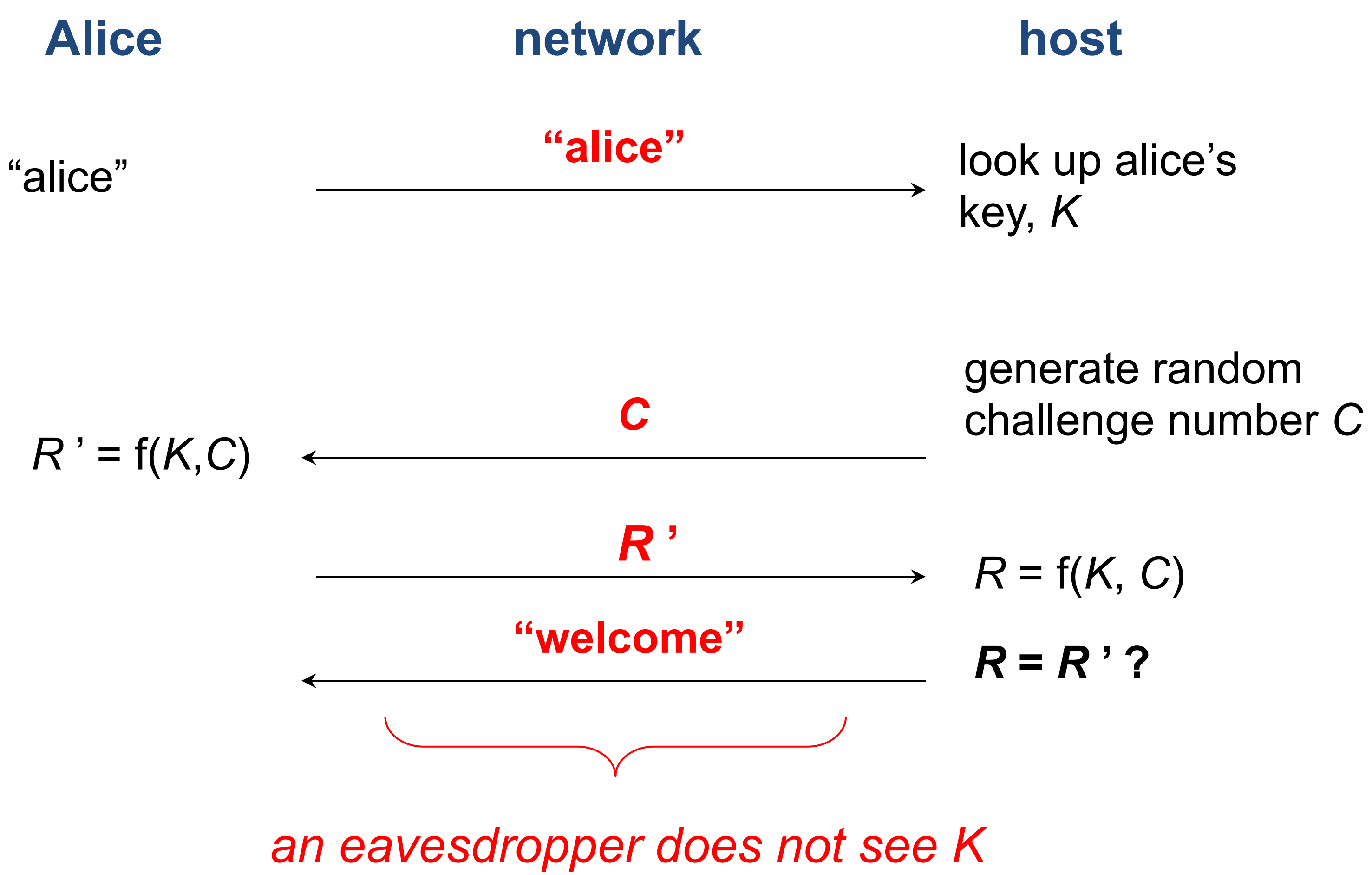


The challenge is a *nonce* (random bits).

We create a hash of the nonce and the secret.

An intruder does not have the secret and cannot do this!

CHAP authentication



SMS/Email Authentication

- Second factor = your possession of a phone (or computer)
- After login, sever sends you a code via SMS (or email)
- Entering it is proof that you could receive the message
- Dangers
  - SIM swapping attacks (social engineering on the phone company)
    - Viable for high-value targets
  - Social engineering to get email credentials

Time-Based Authentication

Time-based One-time Password (TOTP) algorithm

- Both sides share a secret key
  - Sometimes sent via a QR code so the user can scan it into the TOTP app
- User runs TOTP function to generate a one-time password

  one_time_password = *hash*(secret_key, time)

- User logs in with:
  - *Name*, *password*, and *one_time_password*
- Service generates the same password

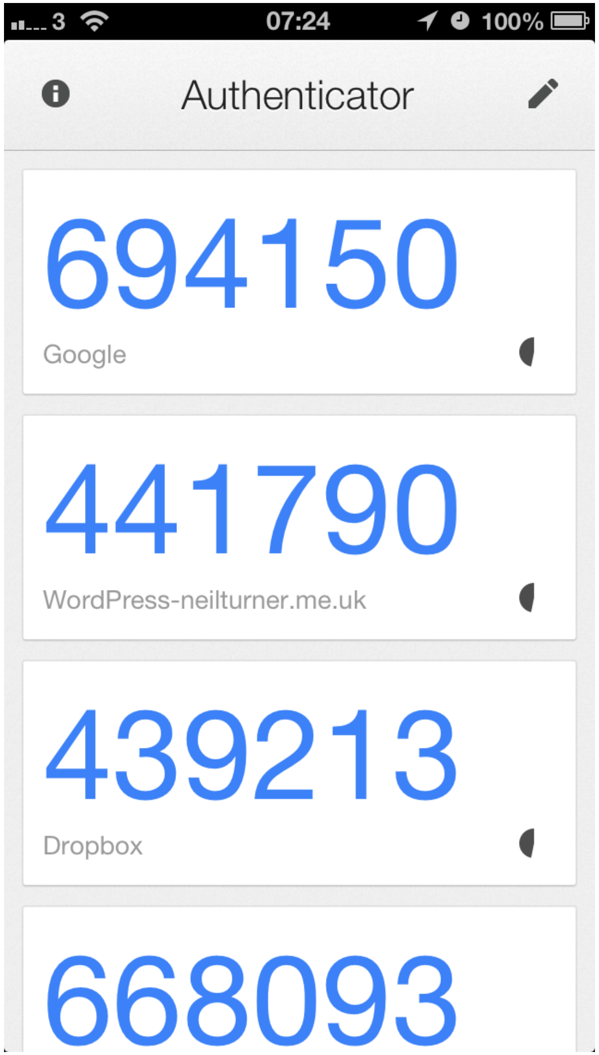
  one_time_password = *hash*(secret_key, time)

- Typically 30-second granularity for time

Time-based One-time Passwords

Used by

- Microsoft Two-step Verification
- Google Authenticator
- Facebook Code Generator
- Amazon Web Services
- Bitbucket
- Dropbox
- Evernote
- Zoho
- Wordpress
- 1Password
- Many others…



RSA SecurID card

Username:

`paul`

Password:

`1234032848`

PIN + passcode from card

Something you know

Something you have

Passcode changes every 60 seconds

1. Enter PIN
2. Press ◊
3. Card computes password
4. Read password & enter

Password:

`354982`

SecurID card

Same principle as Time-based One-Time Passwords

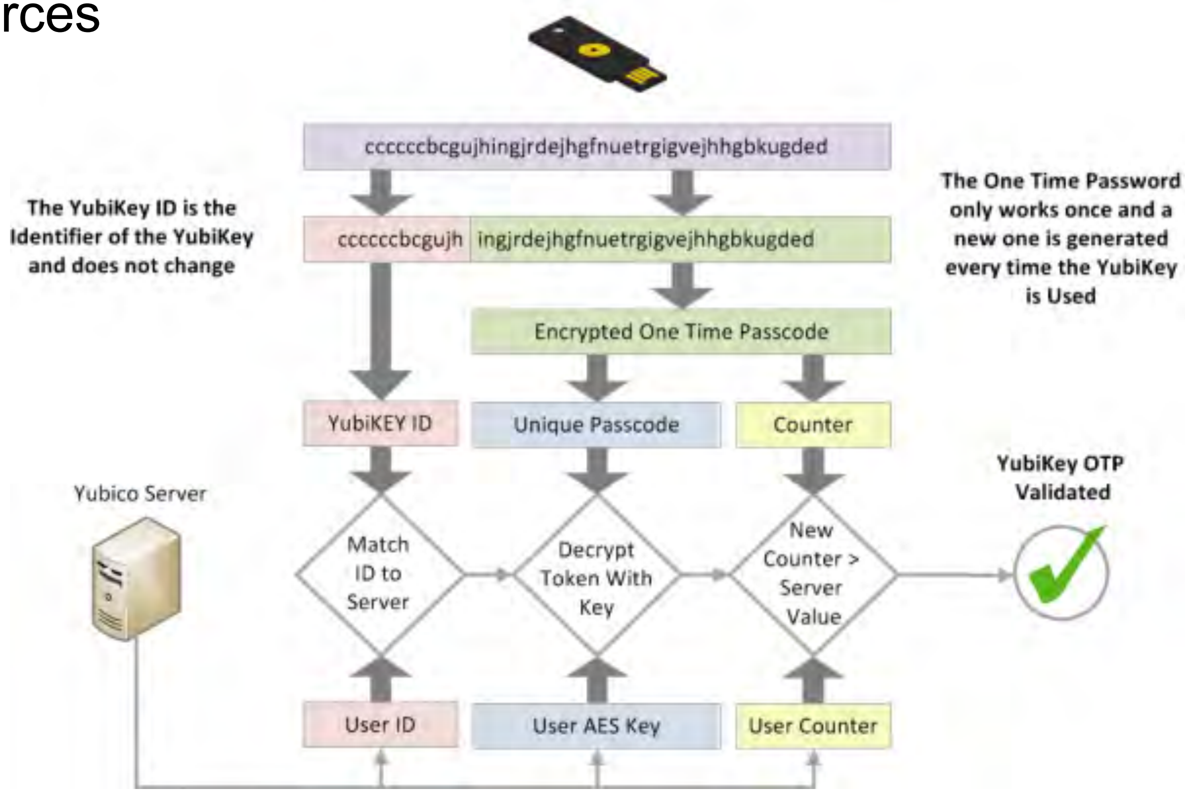
- Proprietary device from RSA
  - SASL mechanism: RFC 2808
- Two-factor authentication based on:
  - **Shared secret key** (seed) ← Something you have
    - stored on authentication card
  - **Shared personal ID** – PIN ← Something you know
    - known by user

Yubikey: Yubico One Time Password

HOTP = Hash-based One-Time Password

OTP = *f*(hardware_id, passcode, counter)

Passcode generated on the device from session counters, previous values, other sources



Man-in-the-Middle Attacks

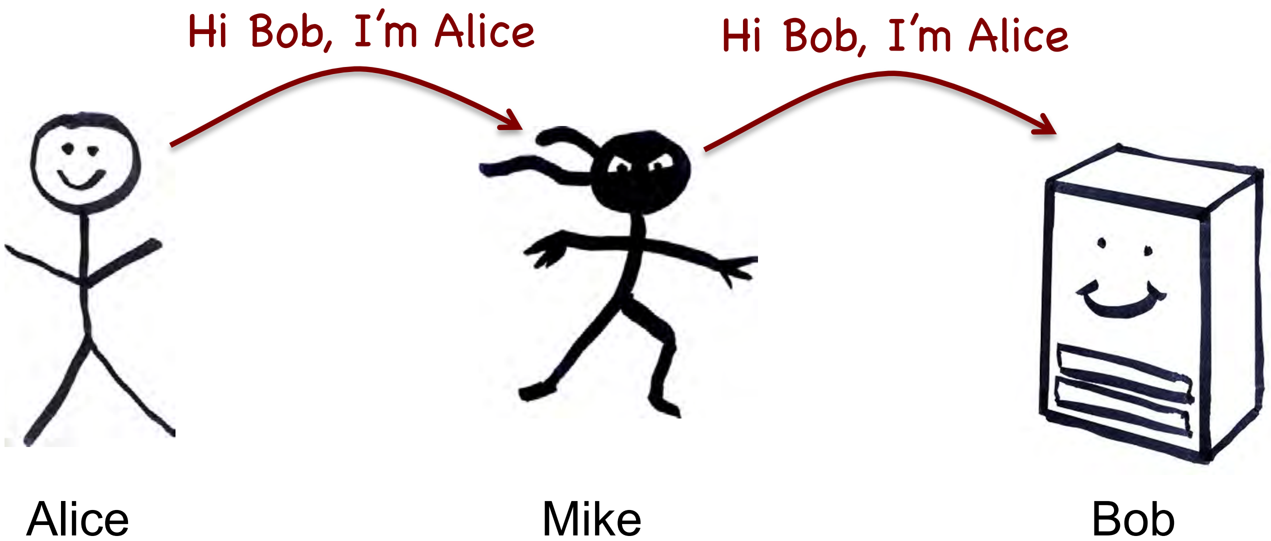
Password systems are vulnerable to man-in-the-middle attacks

- Attacker acts as the server

Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks

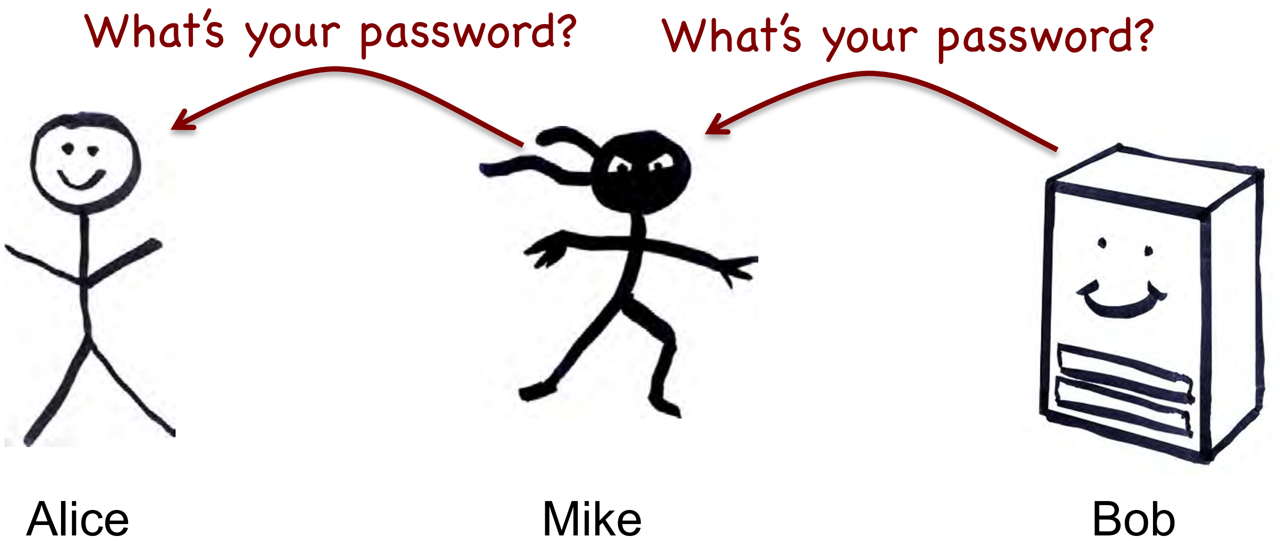
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks

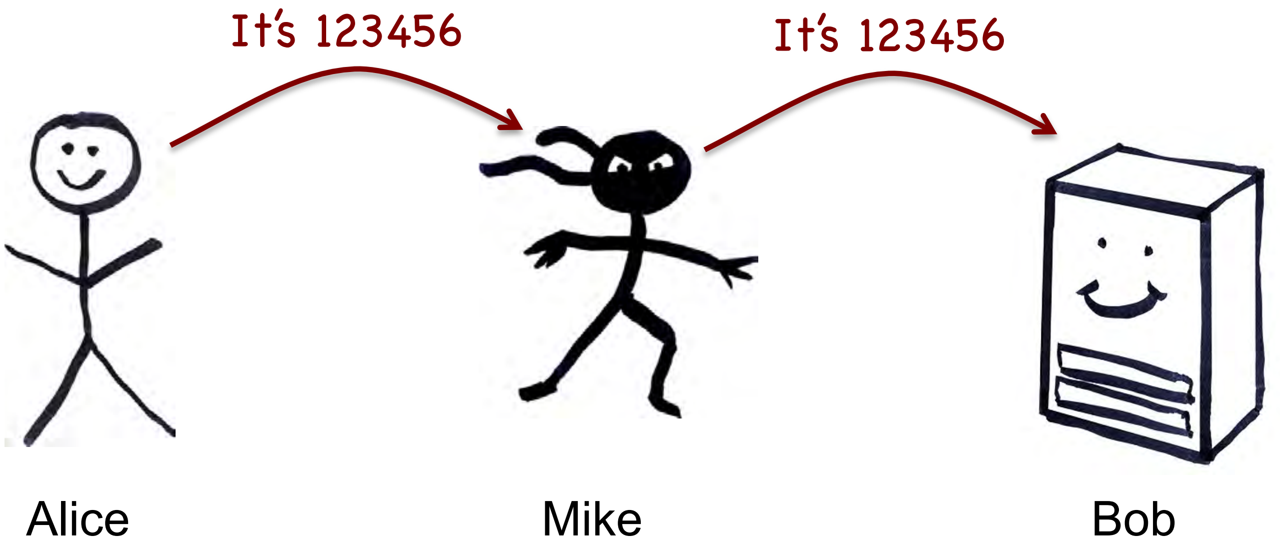
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks

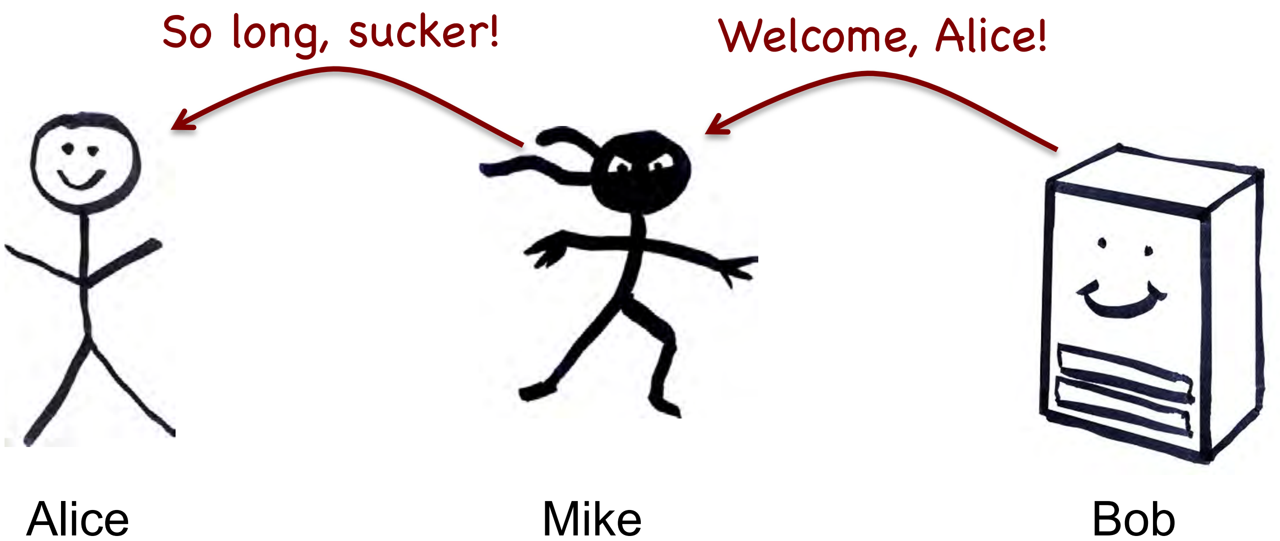
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks

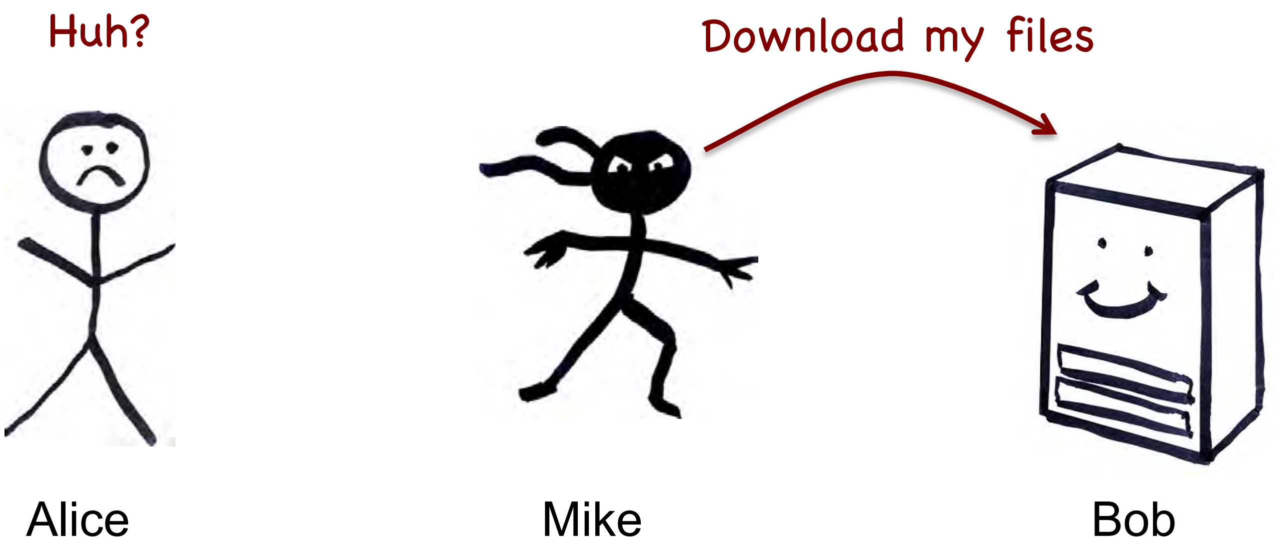
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks

– Attacker acts as the server



Guarding against man-in-the-middle attacks

- Use a covert communication channel
  - The intruder won't have the key
  - Can't see the contents of any messages
  - But you can't send the key over that channel!
- Use signed messages for all communication
  - Signed message = { message, encrypted hash of message }
  - Both parties can reject unauthenticated messages
  - The intruder cannot modify the messages
    - Signatures will fail (they will need to know how to encrypt the hash)
- But watch out for replay attacks!
  - May need to use session numbers or timestamps

The End