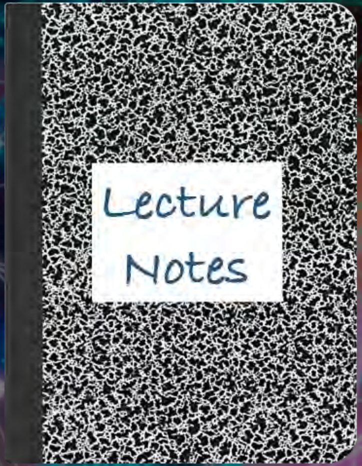
CS 419: Computer Security

Week 4: Authentication



Paul Krzyzanowski

Authentication

Identification: ***who are you?***

Authentication: ***prove it***

Authorization: ***you can do this***

Some protocols (or services) combine all three.

Cryptographic Authentication

Key concept: prove you know a secret (have the key)

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

Alice

Bob

Create a nonce, ***n***
(random bunch of bits)

n →

← $E_K(n)$

Encrypt the nonce with the shared key, ***K***

Validate the result:
$D_K(E_K(n)) \stackrel{?}{=} n$

Bob proved he can do something that requires knowing 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, which is a **long-term key**.

Mutual authentication

- **Alice had Bob prove he has the key**
- **Bob may want to validate Alice as well**

  ⇒ mutual authentication

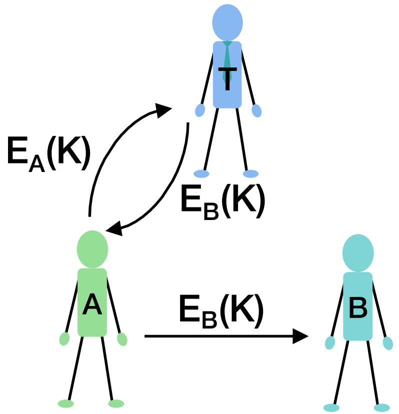
  – Bob will do the same thing: have Alice prove she has the key

Combined authentication & key exchange protocols

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, so she asks Trent**
  - Trent generates a session key encrypted for Alice.
  - Trent encrypts the same key for Bob (called a ***ticket***).
  - Alice can't decrypt the ticket but can send it to Bob.
- **Authentication is implicit:**
  - If Alice can encrypt a message for Trent, she has proved she knows her key
  - If Bob can decrypt the message from Trent, he has proved he knows his key..
- **Trent can also perform *authorization* and reject Alice's request**
- **Weaknesses that we need to address:**
  - Replay attacks

Security Protocol Notation

$Z \,||\, W$

- Z concatenated with W

$A \rightarrow B : \{ Z \,||\, W \} k_{A,B}$

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

$A \rightarrow B : \{ Z \} k_A \;||\; \{ W \} k_{A,B}$

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

r_1, r_2

- nonces – strings of random bits (stands for "*number used once*")

Bootstrap problem

How can 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 … in the context of authentication & key exchange

Simple Protocol

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

Trent creates a session key for Alice and Bob

Alice $\xrightarrow{\text{Request session key to Bob}}$ Trent

Alice $\xleftarrow{\{ k_{A,B} \} k_A \,\|\, \{ k_{A,B} \} k_B}$ Trent

Alice $\xrightarrow{\{ k_{A,B} \} k_B}$ Bob

Alice $\xleftrightarrow{\{ m \} k_{A,B}}$ 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 k_A
  - Trent can **authorize** Alice's request
  - Bob gets a session key encrypted with Bob's key, which only Trent could have created
    - But Bob doesn't know who requested the session – *is the request really from Alice?*
    - *Trent would need to add sender information to the message encrypted for Bob*

- **Vulnerable to replay attacks**
  - Eve records the message from Alice to Bob and later replays it
  - Bob will think he's talking to Alice and will re-use the same session key

- **Protocols should provide authentication & defense against replay attacks**

Needham-Schroeder

Add *nonces* – random strings (r_1, r_2) – to avoid replay attacks

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

② Alice $\xleftarrow{\{\ \text{Alice} \,\|\, \text{Bob} \,\|\, r_1 \,\|\, k_{A,B} \,\|\, \{\ \text{Alice} \,\|\, k_{A,B}\ \}k_B\ \}k_A}$ Trent

③ Alice $\xrightarrow{\{\ \text{Alice} \,\|\, k_{A,B}\ \}k_B}$ Bob

④ Alice $\xleftarrow{\{\ r_2\ \}k_{A,B}}$ Bob

⑤ Alice $\xrightarrow{\{\ r_2 - 1\ \}k_{A,B}}$ Bob

Needham-Schroeder

Add ***nonces*** – random strings – a

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. → Trent: $\{ \text{Alice} \,\|\, \text{Bob} \,\|\, r_1 \}$
2. Alice ← Trent: $\{ \text{Alice} \,\|\, \text{Bob} \,\|\, r_1 \,\|\, k_{A,B} \,\|\, \{ \text{Alice} \,\|\, k_{A,B} \}\, k_B \}\, k_A$
3. Alice → Bob: $\{ \text{Alice} \,\|\, k_{A,B} \}\, k_B$

- Bob now tries to find out if this is a replay attack
- If it is, Eve will not be able to decipher r_2

This is an **authentication** step: Bob asks Alice to prove she knows $k_{A,B}$

4. Alice ← Bob: $\{ r_2 \}\, k_{A,B}$
5. Alice → Bob: $\{ r_2 - 1 \}\, k_{A,B}$

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 obtains the session key from an <u>old</u> message (she worked hard, got lucky, and cracked an earlier message)

Bob sees this as a legitimate request approved by Trent. It was … but earlier!

Replay

③ Eve $\xrightarrow{\{\text{Alice} \,\|\, k_{A,B}\}\, k_B}$ Bob

④ Eve $\xleftarrow{\{r_2\}\, k_{A,B}}$ Bob

⑤ Eve $\xrightarrow{\{r_2 - 1\}\, k_{A,B}}$ Bob

Eve the eavesdropper. She decrypted an <u>old</u> session key and is trying to get Bob to use it to think he's talking to Alice.

Denning-Sacco Solution to the 3rd Message Replay

- **Problem: replay in the third step of the protocol**
  - Eve has an old session key and replays the message: $\{ \text{Alice} \,||\, k_{A,B} \} k_B$
- **Solution: use a timestamp *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
  - Eve cannot spoof a valid message that is encrypted for Bob

Needham-Schroeder w/Denning-Sacco mods

Use nonces – random strings – AND a timestamp

Alice $\xrightarrow{\{\, \text{Alice} \,\|\, \text{Bob} \,\|\, r_1 \,\}}$ Trent

Alice $\xleftarrow{\{\, \text{Alice} \,\|\, \text{Bob} \,\|\, r_1 \,\|\, k_{A,B} \,\|\, \{\, \text{Alice} \,\|\, T \,\|\, k_{A,B} \,\}\, k_B \,\}\, k_A}$ Trent

Alice $\xrightarrow{\{\, \text{Alice} \,\|\, T \,\|\, k_{A,B} \,\}\, k_B}$ Bob

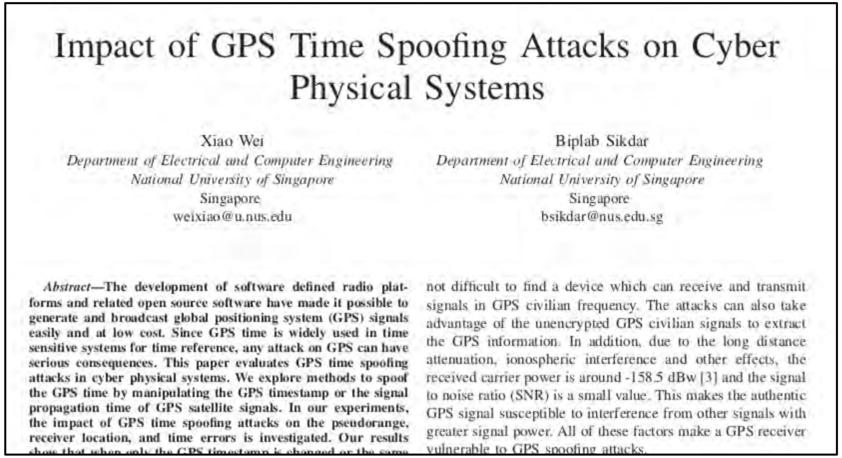
Alice $\xleftarrow{\{\, r_2 \,\}\, k_{A,B}}$ Bob

Alice $\xrightarrow{\{\, r_2 - 1 \,\}\, k_{A,B}}$ 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



Impact of GPS Time Spoofing Attacks on Cyber Physical Systems

Xiao Wei
Department of Electrical and Computer Engineering
National University of Singapore
Singapore
weixiao@u.nus.edu

Biplab Sikdar
Department of Electrical and Computer Engineering
National University of Singapore
Singapore
bsikdar@nus.edu.sg

***Abstract*—The development of software defined radio platforms and related open source software have made it possible to generate and broadcast global positioning system (GPS) signals easily and at low cost. Since GPS time is widely used in time sensitive systems for time reference, any attack on GPS can have serious consequences. This paper evaluates GPS time spoofing attacks in cyber physical systems. We explore methods to spoof the GPS time by manipulating the GPS timestamp or the signal propagation time of GPS satellite signals. In our experiments, the impact of GPS time spoofing attacks on the pseudorange, receiver location, and time errors is investigated. Our results**

not difficult to find a device which can receive and transmit signals in GPS civilian frequency. The attacks can also take advantage of the unencrypted GPS civilian signals to extract the GPS information. In addition, due to the long distance attenuation, ionospheric interference and other effects, the received carrier power is around -158.5 dBw [3] and the signal to noise ratio (SNR) is a small value. This makes the authentic GPS signal susceptible to interference from other signals with greater signal power. All of these factors make a GPS receiver vulnerable to GPS spoofing attacks.

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
  - This is a **session identifier** & included in all messages for the session
- **This is a slightly different protocol (a form of challenge-response)**
  - 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)

Alice $\longrightarrow$ Bob: $n \,\|\, \text{Alice} \,\|\, \text{Bob} \,\|\, \{r_1 \,\|\, n \,\|\, \text{Alice} \,\|\, \text{Bob}\}\, k_A$

Alice sends the communication request to Bob – with the session ID

Trent $\longleftarrow$ Bob: $n \,\|\, \text{Alice} \,\|\, \text{Bob} \,\|\, \{r_1 \,\|\, n \,\|\, \text{Alice} \,\|\, \text{Bob}\}\, k_A$

$\{r_2 \,\|\, n \,\|\, \text{Alice} \,\|\, \text{Bob}\}\, k_B$

Bob authenticates himself & forwards the request to Trent

Trent $\longrightarrow$ Bob: $n \,\|\, \{r_1 \,\|\, k_{A,B}\}\, k_A \,\|\, \{r_2 \,\|\, k_{A,B}\}\, k_B$

Trent sends Bob the session key for both Alice & Bob

Alice $\longleftarrow$ Bob: $n \,\|\, \{r_1 \,\|\, k_{A,B}\}\, k_A$

Alice is asked to prove that she can extract the key and use r_1

Kerberos

Kerberos

- **Authentication service developed by MIT**
  - Created as part of Project Athena 1983-1988

- **Uses a trusted third party & symmetric cryptography**

- **Based on *Needham Schroeder* with the *Denning Sacco* modification**

- **Passwords are never sent in clear text**
  - Assumes only the network can be compromised

- **Supported in most all popular operating systems**
  - Default network authentication used in Microsoft Windows
  - Supported in macOS, Linux, FreeBSD, z/OS, …
  - Used by Rutgers LCSR to manage NetIDs via the Central Authentication Service (CAS)

Kerberos

Users and services authenticate themselves to each other

To access a service:

- User presents a ticket issued by the Kerberos authentication server
- Service uses 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 distribution: giving both parties an encryption key (securely)

Kerberos – General Flow

User *Alice* wants to communicate with a service *Bob*

Both Alice and Bob have keys – Kerberos has copies

– key = *hash*(password)

Step 1:

– Alice authenticates with Kerberos server
  - Gets *session key* and *ticket*

Step 2:

– Alice gives Bob the ticket, which contains the session key
– Convinces Bob that she got the session key from Kerberos

Kerberos (1): Authorize, Authenticate

Alice

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

{ "Alice" || "Bob" } →

Kerberos Authentication Service (AS)

If Alice is allowed to talk to Bob, generate session key, $k_{A,B}$

← { "Bob's server", T, $k_{A,B}$ } k_A

Alice decrypts this:

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

TICKET

← { "Alice", T, $k_{A,B}$ } k_B

eh? (Alice can't read this!)

Kerberos (2): Send key

Alice

Alice encrypts a timestamp with session key

$$\{ \text{"Alice"}, k_{A,B} \}\, k_B \;||\; \{ T' \}\, k_{A,B}$$

ticket

Bob

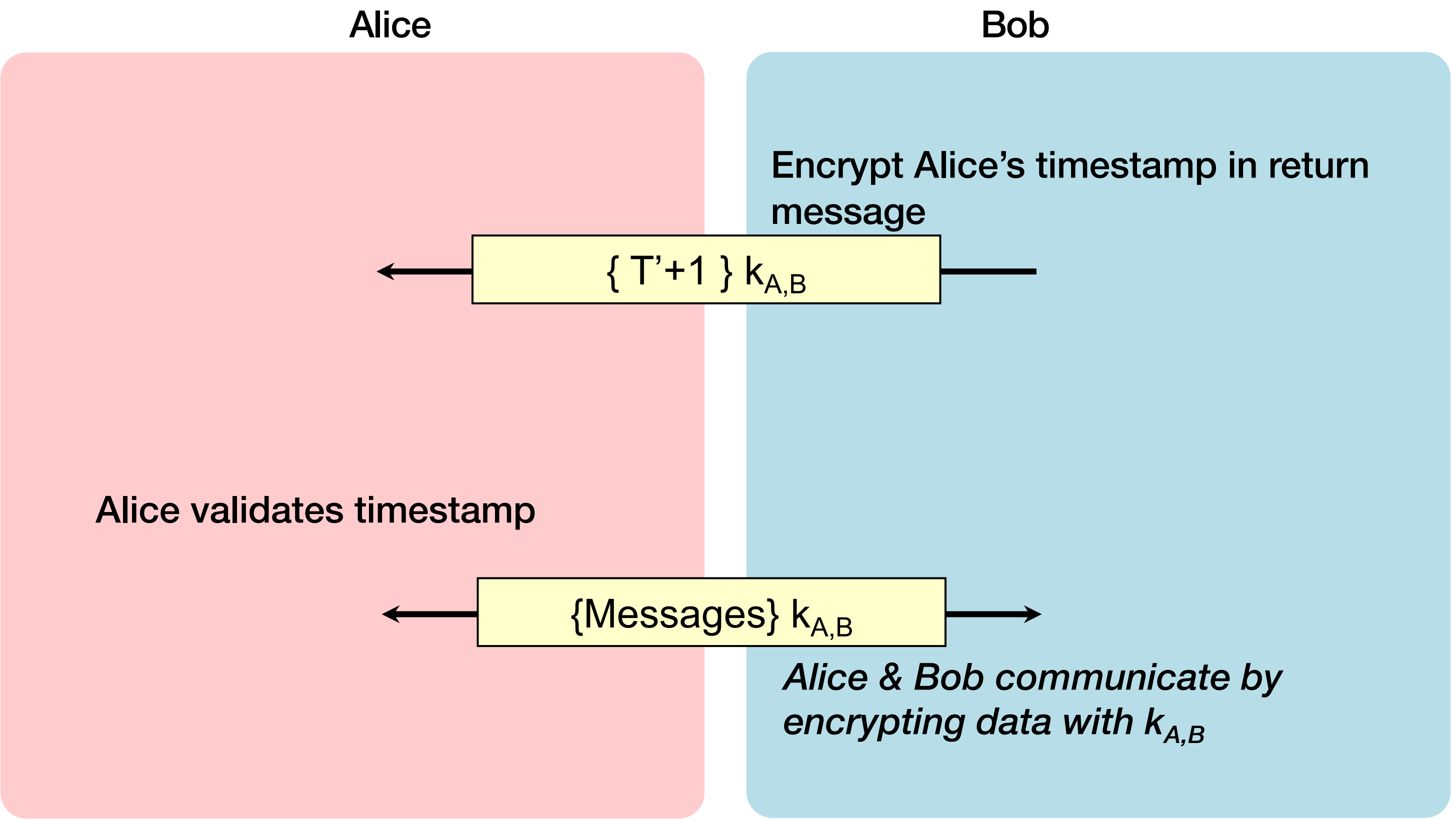
Bob decrypts the ticket:

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

Decrypts time stamp

- Validates time window
- Prevents replay attacks

Kerberos (3): Authenticate recipient of message



Kerberos key usage

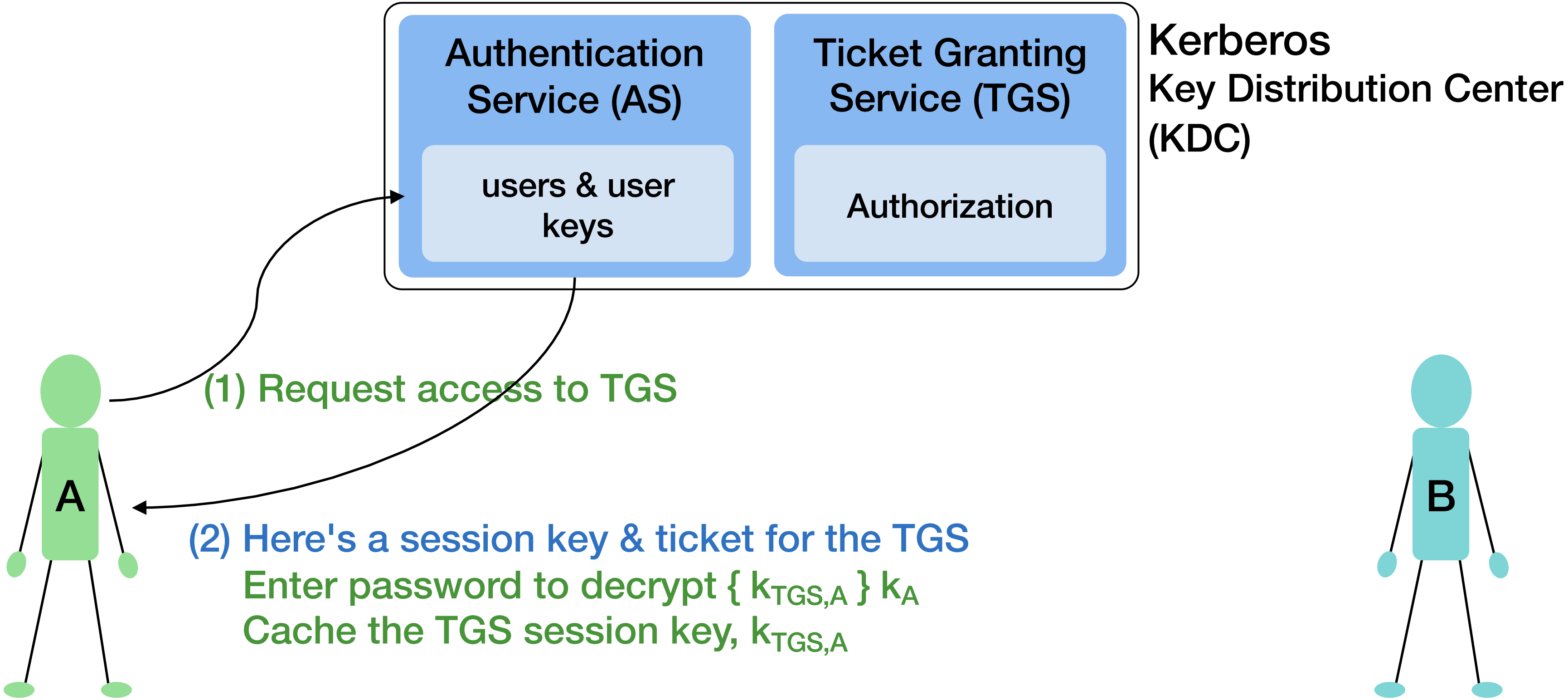
- **Every time a user wants to access a service**
  - User's password (key) must be used to decode the message from Kerberos
- **We can avoid this by caching the password in a file**
  - Not a good idea
- **Another way: create a temporary password**
  - We can cache this temporary password
  - It's just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

Authentication Service + Ticket Granting Service

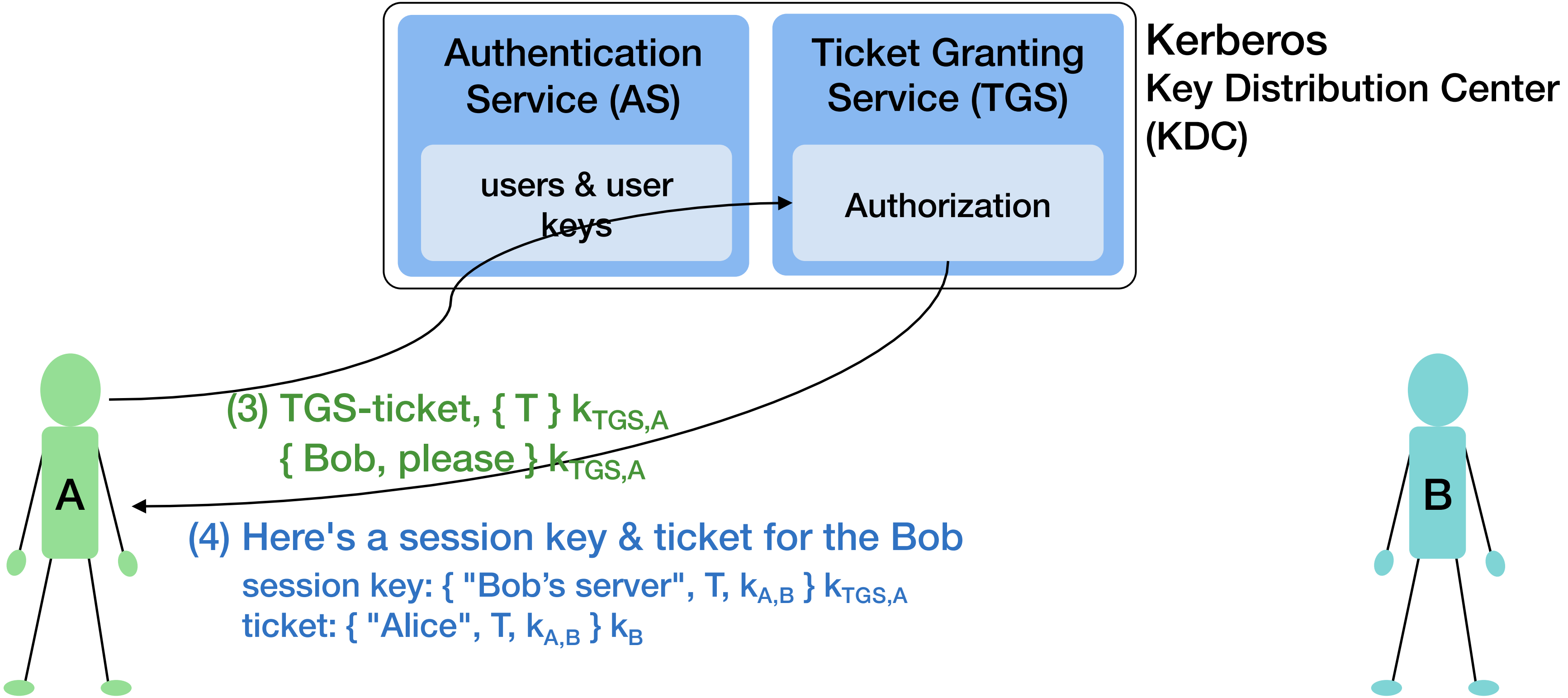
Ticket Granting Server (TGS)

- **TGS works like a temporary ID**
- **User first requests access to the TGS**
  - Contact Kerberos Authentication Service (AS knows all users & their keys)
    - 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

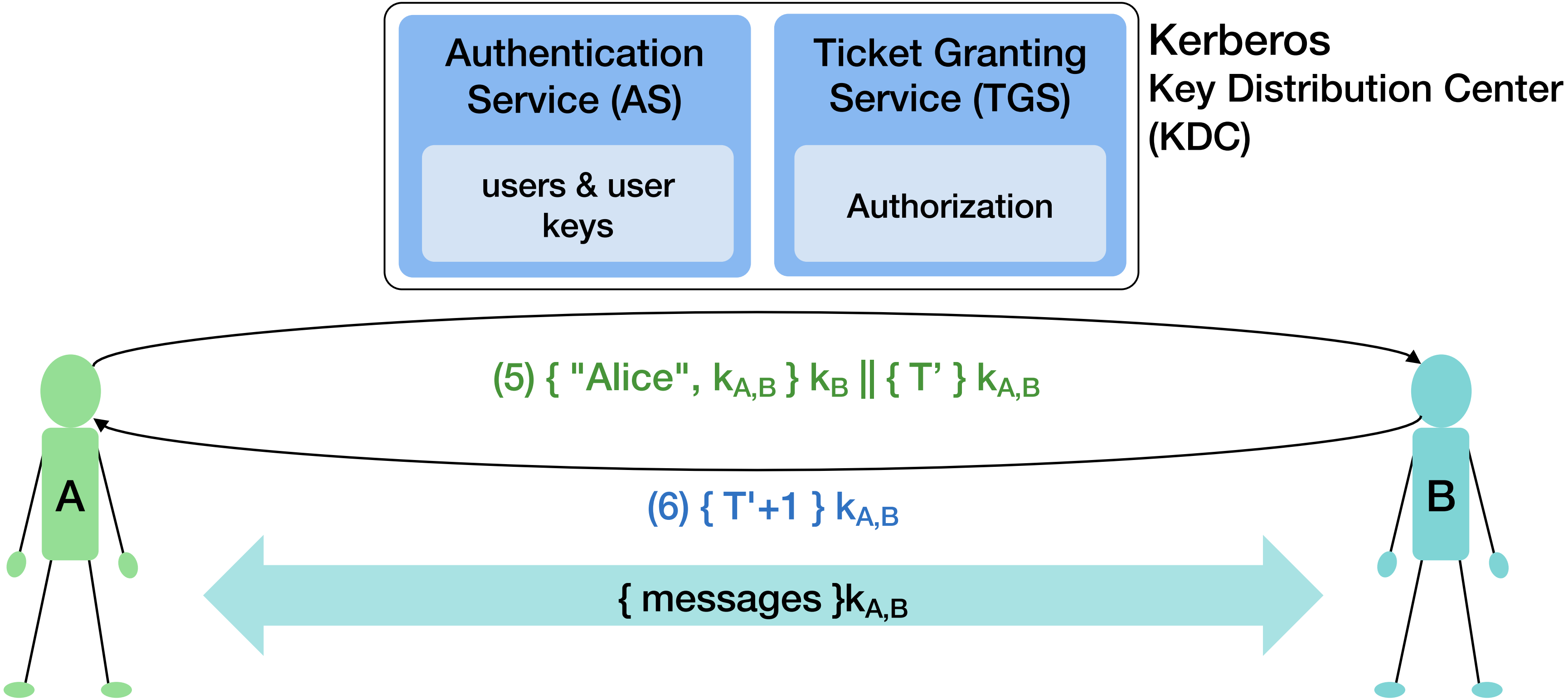
Kerberos AS + TGS: Steps 1 & 2 – Get the TGS Key



Kerberos AS + TGS: Steps 3 & 4: Get Key for Bob



Kerberos AS + TGS: Step 5: Give Bob the Key



Using Kerberos

`$ kinit`

`Password:` *enter password*

ask AS for permission (session key) to access TGS

Alice gets:

{"TGS", T, $k_{A,TGS}$ } k_A ← *Session key & encrypted timestamp*

{"Alice", $k_{A,TGS}$ } k_{TGS} ← *TGS Ticket*

Compute key (A) from password to decrypt session key $k_{A,TGS}$ 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 → TGS

$\{\text{“Alice”}, k_{A,TGS}\}\, k_{TGS}, \{T\}\, k_{A,TGS}$

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

$\{\text{“rlogin@somehost”}, k_{A,R}\}\, k_{A,TGS}$

$\{\text{“Alice”}, k_{A,R}\}\, k_R$

$k_{A,R}$ = session key for *rlogin*

ticket for rlogin server on somehost

Summary: Combined authentication & key exchange

Basic idea with symmetric cryptography:

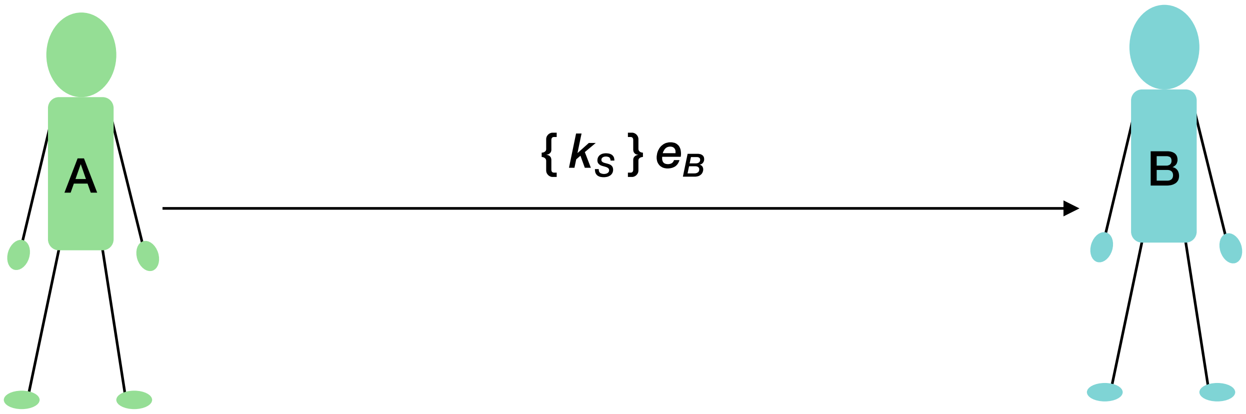
Use a trusted third party (Trent) that has all the keys

- **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

Public Key Based Key Exchange

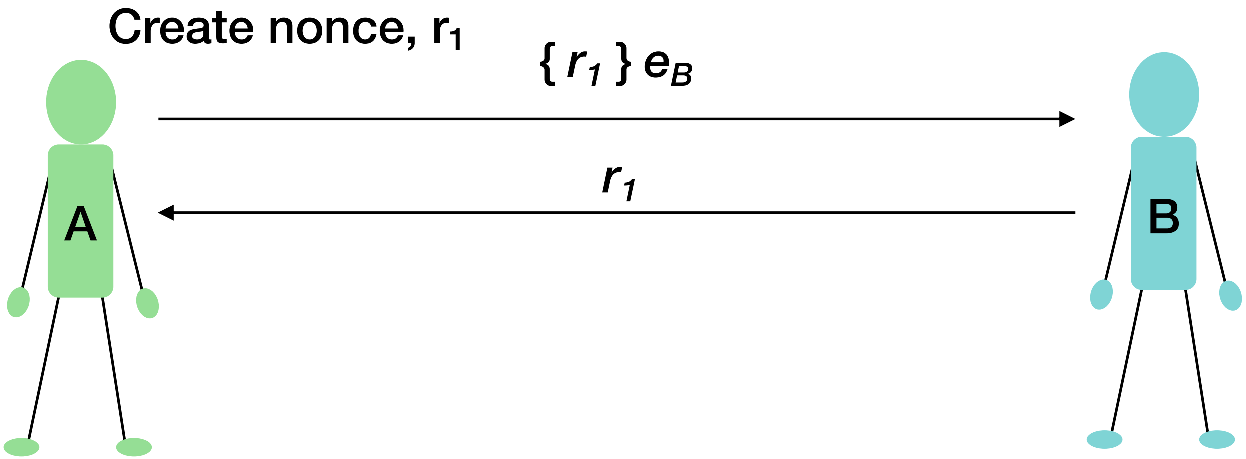
We saw how this works…

- 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:



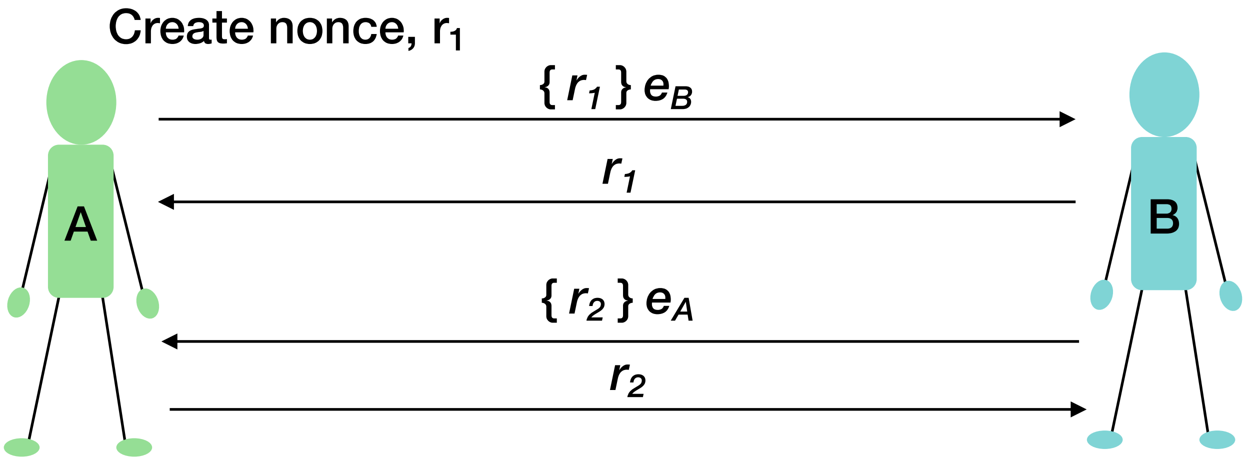
Adding authentication

- **Have Bob prove that he has the private key**
  - Same way as with symmetric cryptography – prove he can encrypt or decrypt



Adding mutual authentication

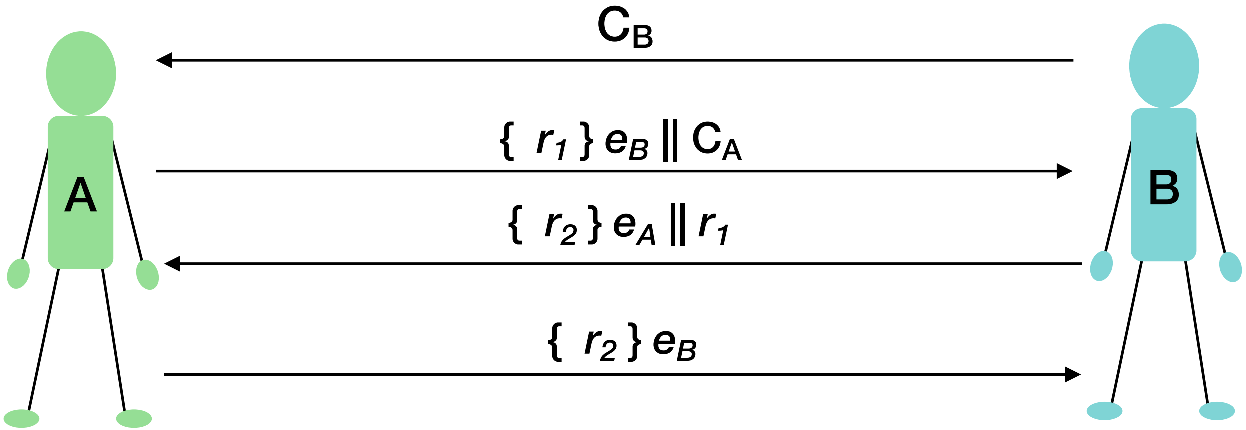
- **Bob asks Alice to prove that she has her private key**



Adding identity binding

- **How do we know we have the right public keys?**
- **Get the public key from a trusted certificate**
  - Validate the signature on the certificate before trusting the public key within

Note: This does not provide forward security – we'll look at that later.



Cryptographic toolbox

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

Authentication protocols without key exchange

Three Factors of Authentication

1. Ownership **Something you have**	*Key, card*	*Can be stolen*
2. Knowledge **Something you know**	*Passwords, PINs*	*Can be guessed, shared, stolen*
3. Inherence **Something you are**	*Biometrics (face, fingerprints)*	*Requires hardware* *May be copied* *Not replaceable if lost or stolen*

Multi-Factor Authentication (MFA)

Factors may be combined

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

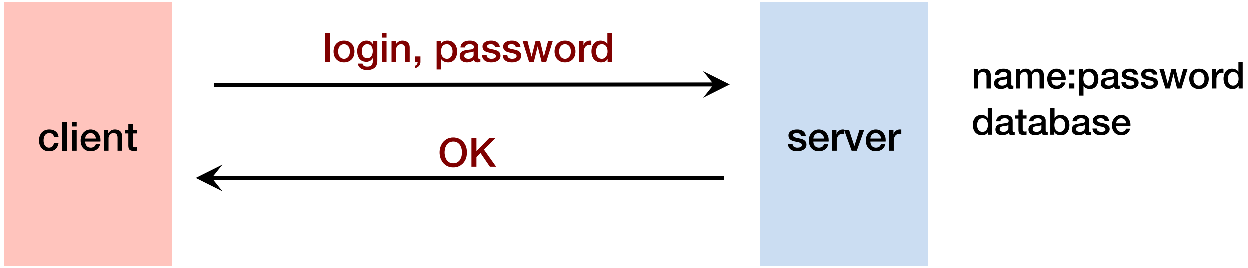
- **Password + code delivered via SMS: 2-factor authentication**
  - Password — something you know
  - Code — something you have: your phone

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 username (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

It's not getting better

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

Rank	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024
1	123456	123456	123456	123456	123456	123456	123456	password	123456	123456
2	password	password	password	password	123456789	123456789	123456789	123456	admin	123456789
3	12345678	12345	12345678	123456789	qwerty	picture1	12345	123456789	12345678	12345678
4	qwerty	12345678	qwerty	12345678	password	password	qwerty	guest	123456789	password
5	12345	football	12345	12345	1234567	12345678	password	qwerty	1234	qwerty123
6	123456789	qwerty	123456789	111111	12345678	111111	12345678	12345678	12345	qwerty1
7	football	1234567890	letmein	1234567	12345	123123	111111	111111	password	111111
8	1234	1234567	1234567	sunshine	iloveyou	12345	123123	12345	123	12345

Top passwords by year 2015-2019: SplashData; 2020-2023: NordPass

https://nordpass.com/most-common-passwords-list/

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

Policies to the rescue?

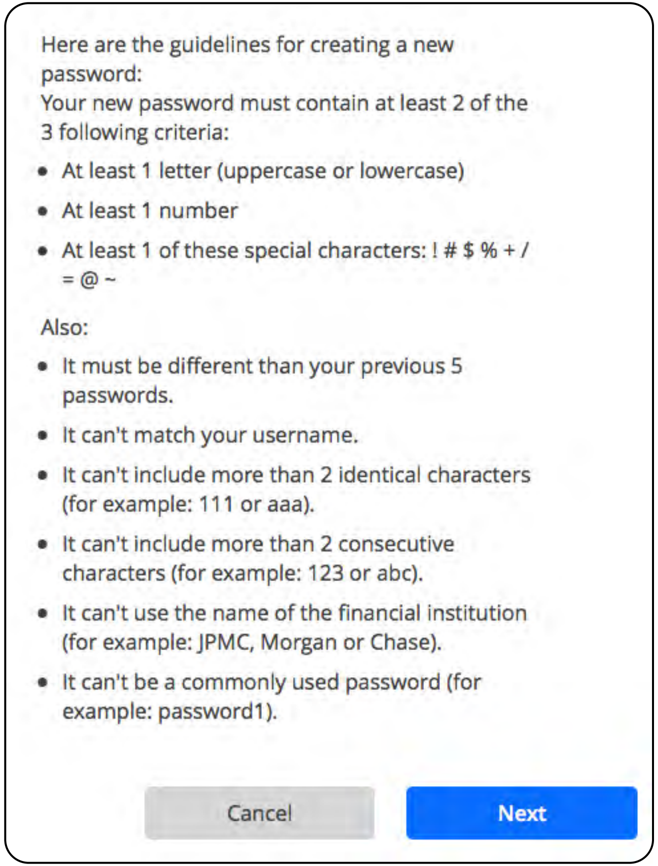
Password rules

> *“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 requirement problems

- 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 – 28 Aug 2024 Draft

- **Do not:**
  - Require periodic password changes
  - Impose composition or complexity requirements (certain # of numbers, letters, symbols)
  - Require passwords to be at least 8 characters long
  - Store a password hint that is accessible to others
  - Use knowledge-based authentication (KBA) ("*what was the name of your pet?*")
  - Validate a truncated version of the password
  - Reuse recent passwords
- **Prefer**
  - Passwords should be a minimum of 15 characters long, support at least 64 chars
  - Unicode and ASCII should be permitted
- **Avoid**
  - Passwords obtained from databases of previous breaches
  - Dictionary words and common phrases
  - Repetitive or sequential characters (e.g. ‘`aaaaa`’, ‘`1234abcd`’)
  - Context-specific words, such as the name of the service, the username, and derivatives

NIST Special Publication
NIST SP 800-63-4 2pd
Digital Identity Guidelines
Second Public Draft

David Temoshok
Ryan Galluzzo
Connie LaSalle
Naomi Lefkovitz
Applied Cybersecurity Division
Information Technology Laboratory

Andrew Regenscheid
Computer Security Division
Information Technology Laboratory

Yee-Yin Choong
Information Access Division
Information Technology Laboratory

Diana Proud-Madruga
Sarbari Gupta
Electrosoft

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-63-4.2pd

August 2024

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

https://pages.nist.gov/800-63-4/sp800-63b.html

https://arstechnica.com/security/2024/09/nist-proposes-barring-some-of-the-most-nonsensical-password-rules/

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 an 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

Poor Password Management

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!

	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
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

Top 26 Adobe Passwords

Meta stored 600 million Facebook and Instagram passwords in plain text

William Gallagher • September 27, 2024

Across Facebook and Instagram, Meta has been storing more than half a billion users' passwords in plain text, with some easily readable for more than a decade.

The issue was first uncovered in 2019 when Facebook admitted to "hundreds of millions" of passwords being stored unencrypted. Facebook, now Meta, said that the passwords were not available outside of the company — but also admitted that around 2,000 engineers had made about 9 million queries on that user database.

Now Meta's operation in Ireland has finally been fined $101.5 million after a five-year investigation by the Irish Data Protection Commission (DPC). The fine is levied under Europe's stringent General Data Protection Regulation (GDPR).

"It is widely accepted that user passwords should not be stored in plaintext, considering the risks of abuse that arise from persons accessing such data," said Graham Doyle, Deputy Commissioner at the DPC, in a statement about the fine. "It must be borne in mind, that the passwords the subject of consideration in this case, are particularly sensitive, as they would enable access to users' social media accounts."

https://appleinsider.com/articles/24/09/27/meta-stored-600-million-facebook-and-instagram-passwords-in-plain-text

PAP: Reusable passwords

Solution:

Store a hash of the password in a file

- Given a file, you don't get the passwords, only their hashes
  - Hashes are one-way functions
  - Example, Linux passwords hashed with a SHA-512 hash (SHA-2)
- Have to resort to a dictionary or brute-force attack

Dictionary attack vs. Brute force

- **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 10 billion passwords per second: ≈ 6 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

TIME IT TAKES A HACKER TO BRUTE FORCE YOUR PASSWORD IN 2024

Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	3 secs	6 secs	9 secs
5	Instantly	4 secs	2 mins	6 mins	10 mins
6	Instantly	2 mins	2 hours	6 hours	12 hours
7	4 secs	50 mins	4 days	2 weeks	1 month
8	37 secs	22 hours	8 months	3 years	7 years
9	6 mins	3 weeks	33 years	161 years	479 years
10	1 hour	2 years	1k years	9k years	33k years
11	10 hours	44 years	89k years	618k years	2m years
12	4 days	1k years	4m years	38m years	164m years
13	1 month	29k years	241m years	2bn years	11bn years
14	1 year	766k years	12bn years	147bn years	805bn years
15	12 years	19m years	652bn years	9tn years	56tn years
16	119 years	517m years	33tn years	566tn years	3qd years
17	1k years	13bn years	1qd years	35qd years	276qd years
18	11k years	350bn years	91qd years	2qn years	19qn years

Hardware: 12 x RTX 4090
Password hash: bcrypt

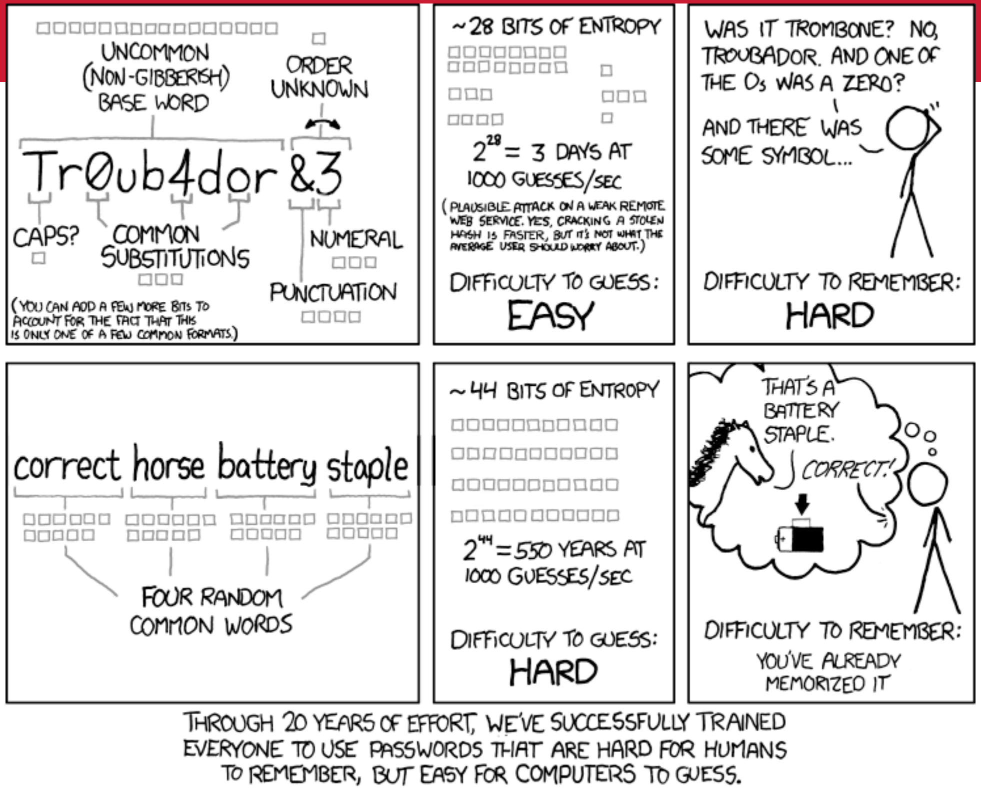
- Note: the benchmarks changed from MD5 to bcrypt. Bcrypt is designed to be slow – about a million times slower than MD5.
- macOS uses SHA-512
- Linux supports different types of hashes and the default depends on the distribution. *yescrypt* is common as a memory-intensive, slow hash that isn't optimized by GPUs.

› Learn more about this at hivesystems.com/password

Longer passwords

English text has an entropy of about 1.2-1.5 bits per character

Random text has an entropy ≈ $\log_2(1/95) \approx 6.6$ bits/character



Assume 95 printable characters

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)

"**`VhsRrsFA`**" + "**`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 and becomes part of the hash)

Example:

SHA-256 hash of "`password`", salt = "`VhsRrsFA`"= *hash*("`VhsRrsFApassword`") =

`b791b1b572c0025ef30ecc5fc5ecc5c623f52fca66250560fce8d22623b166c8`

*You will **not** have a precomputed hash("VhsRrsFApassword")*

Linux example – salted hashes

- **The passwords are both monkey**
- **One has a salt of mysalt123 and the other mysalt124 – one byte off**

```
$ mkpasswd --method=sha-256 --salt=mysalt123 monkey
$5$mysalt123$uw7/eKvgmWOARTME9U2eQIWhO0efP1mPfK9rnXmUBLD

mkpasswd --method=sha-256 --salt=mysalt124 monkey
$5$mysalt124$sBfthw62ybrQg04PEECUBnJFSW6BV5xOV/5hoswQtS/
```

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**
  - OpenBSD *bcrypt* Blowfish password hashing algorithm

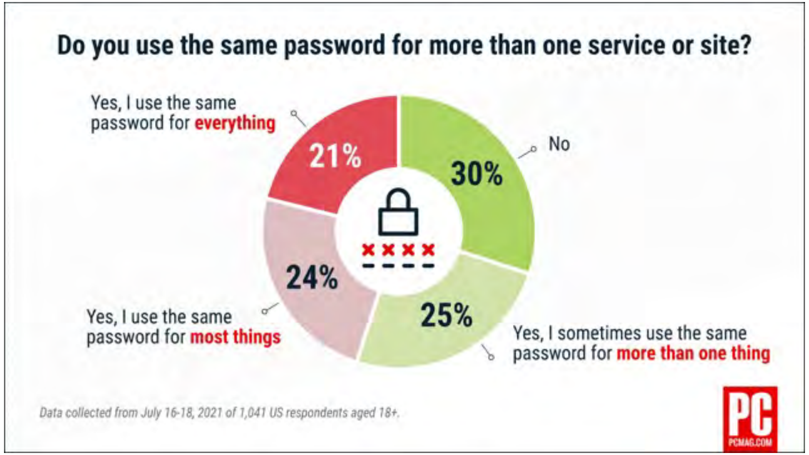
People forget passwords

Especially seldom-used ones. How can we handle that?

Email them?	– Common solution – Requires that the server stores the password (not 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 – What if someone reads your mail, or you no longer have that address?
Provide hints?	– An attacker can get the hints too
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 username
- **This is the root of phishing attacks**



PC Magazine, September 21, 2021

https://www.pcmag.com/news/stop-using-the-same-password-on-multiple-sites-no-really

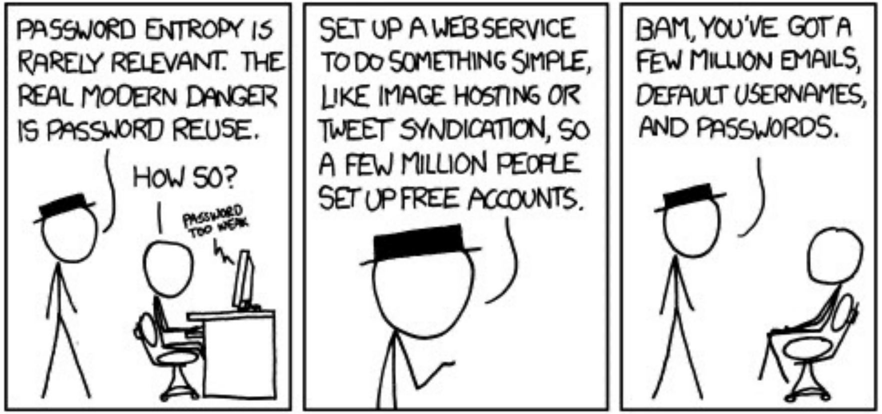
Credential Stuffing & Password Spraying Attacks

- **Credential Stuffing Attack**
  - Assumes people might use the same password on different accounts
  - Get credentials for a user (e.g., buy them on a dark web marketplace)
  - Log in to lots of unrelated accounts trying those credentials

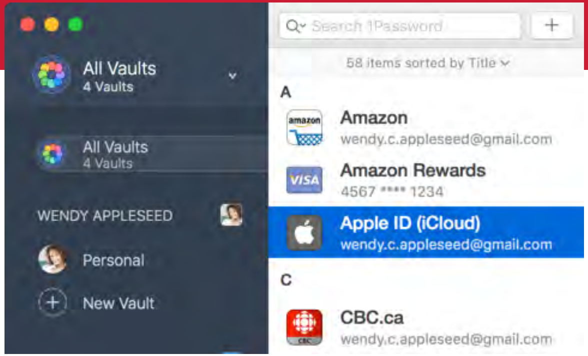
    Example:

    If you got name="`bobsmith1998`", password="`monkey123`" on facebook.com
    the same login credentials might work on paypal.com

- **Password Spraying Attack**
  - Instead of trying multiple guesses for one account, try a common password on a huge number of accounts
  - Avoids lockout and detection from trying too many passwords

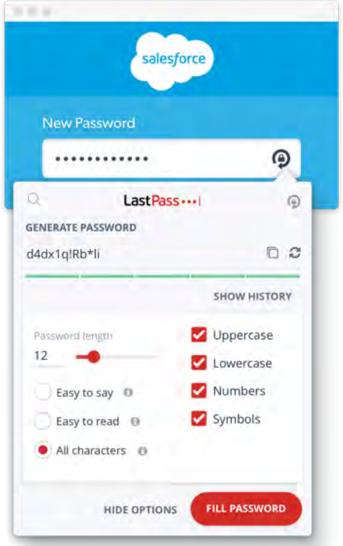


Password Managers



Software that stores passwords in an encrypted file

- **Do you trust the protection?**
  - The reputation of the company & its security policies
  - 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
  - Generates strong passwords
  - Password managers may have the ability to recognize web sites & defend against phishing while providing auto-complete convenience for legitimate sites



9 Popular Password Manager Apps Found Leaking Your Secrets

Tuesday, February 28, 2017 · Wang Wei

Share 7 · Share · Tweet · Share

REPORT
Vulnerabilities in Password Manager Apps

- Dashlane: #1 Password Manager
- F-Secure KEY Password manager
- 1Password - Password Manager
- Password Manager
- My Passwords
- Keeper®: Free Password Manager

THE VERGE

LastPass fixes bug that could let malicious websites extract your last used password

Even password managers have security bugs

By Jon Porte

LastPass ha
malicious we
entered by t
reports that
a researcher

WIRED · SECURITY · POLITICS · GEAR · THE BIG STORY · MORE

LILY HAY NEWMAN · SECURITY · MAR 4, 2023 9:00 AM

Security News This Week: The LastPass Hack Somehow Gets Worse

Plus: The US Marshals disclose a "major" cybersecurity incident, T-Mobile has gotten pwned so much, and more.

The Washington Post

Password managers have a security flaw. But you should still use one.

Exclusive: A new study finds bugs in five of the most popular password managers. So how is it safe to keep all your eggs in one basket?

By Geoffrey A. Fowler • Feb 19, 2019

BLEEPINGCOMPUTER

Bitwarden flaw can let hackers steal passwords using iframes

By **Bill Toulas** · March 8, 2023 · 05:08 PM · 6

CSO

Design flaw has Microsoft Authenticator overwriting MFA accounts, locking users out

By Evan Schuman
August 5, 2024

With use of multi-factor authentication rising, end-users can find themselves fiddling with codes and authentication apps frequently throughout their days. For those who rely on Microsoft Authenticator, the experience can go beyond momentary frustration to full-blown panic as they become locked out of their accounts.

That's because, due to an issue involving which fields it uses, Microsoft Authenticator often overwrites accounts when a user adds a new account via QR scan — the most common method of doing so.

Forbes

Warning As 1Password, DashLane, LastPass And 3 Others Leak Passwords

By Davey Winder
December 11, 2023

Six of the most popular password managers have been called out by security researchers who uncovered a major vulnerability that impacts the Android autofill function. The AutoSpill vulnerability enables hackers to bypass the security mechanisms protecting the autofill functionality on Android devices, exposing credentials to the host app calling for them.

https://www.csoonline.com/article/3480918/design-flaw-has-microsoft-authenticator-overwriting-mfa-accounts-locking-users-out.html

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 http, 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 (doesn't help with shoulder surfing)**

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

(3) **Use one-time passwords**

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) or *f*(secret, sequence#)
2. **Challenge-based**: *f*(challenge, secret)
3. **Time-based**: password = *f*(time, 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 a random number, R
- Using a one-way function (e.g., a hash function), $f(x)$:

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

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 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 *f*(x_{100} provided by alice) = passwd("alice")
    replace x_{101} in db with x_{100} 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}.

S/Key → OPIE

S/Key slightly refined by the U.S. Naval Research Laboratory (NRL)

- **OPIE = One time Passwords In Everything**
  - Comes with FreeBSD, OpenBSD; available on Linux & other POSIX platforms
  - Use /usr/sbin/opielogin instead of standard /bin/login program
- **Same iterative generation as S/Key**

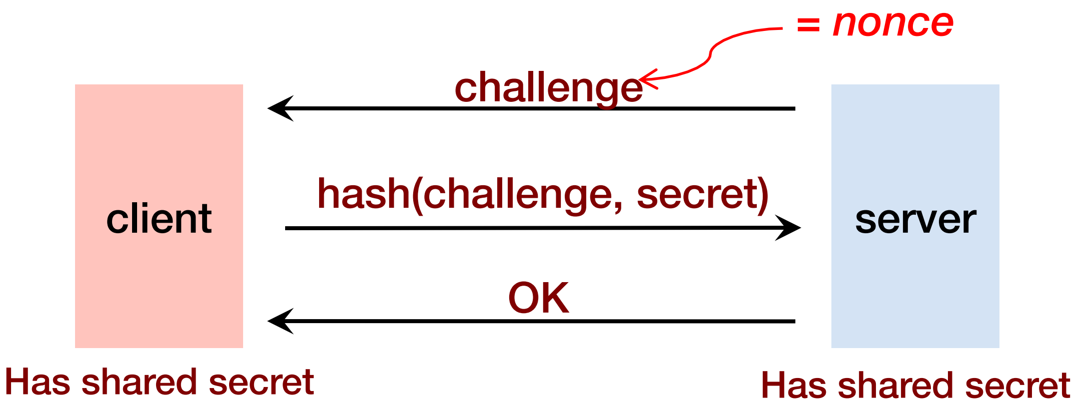
  starting_password = Hash(*seed*, *secret_pass_phrase*)

  The *seed* can differ among applications and enables a user to use the same passphrase securely for different applications
- **Operates in two modes**
  - Sequence-based: pre-generate a sequence of one-time passwords
    - A password is represented as 6 short words
  - Challenge-based: user is presented with a sequence number
    - Computes the proper password from a stored seed value

See http://manpages.ubuntu.com/manpages/bionic/man4/opie.4freebsd.html

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

Alice	network	host
“alice”	“alice” →	look up alice’s key, *K*
	← *C*	generate random challenge number *C*
$R' = f(K,C)$		
	R ’ →	$R = f(K, C)$
	← “welcome”	$R = R'$?

an eavesdropper does not see K

Passkeys - WebAuthn

Passkeys = Passwordless login – endorsed by Apple, Google, Microsoft

- Avoid problems of having users choose strong, unique passwords
- Avoids phishing attacks
- Based on public key cryptography
  - Credentials can be backed up and replicated across user devices

Device generates public/private key pair for a specific service

- Private key is stored locally – the service never sees it
  - Its use can be authorized with Touch ID, Face ID, local device/user password
- Public key is sent to the server – it associates it with the user account

Passkeys – Setup

User Alice | **Service**

username: alice →

Password and/or authentication code →

Conventional login

Create public/private key pair for the service

Elliptic Curve or RSA algorithms

Public key →

Note: the public key is not secret

Store private key securely

Accessible via local password or biometrics.

Associate this key with the service.

Each passkey is unique for each service

Store public key with user info

Enable lookup when presented with a user login name

See https://passage.id/post/what-is-webauth

Passkeys – Login

User Alice — **Service**

username: alice →

← *Here's a challenge:* `XdQLAxB1L1`*…*

Generate signature for challenge using your private key

signature(challenge) →

Authorize access to private key via Touch ID, Face ID, password, …

Validate signature using the user's public key for the service

← *Welcome, Alice!*

See https://passage.id/post/what-is-webauth

TOTP: Time-Based One-Time Passwords

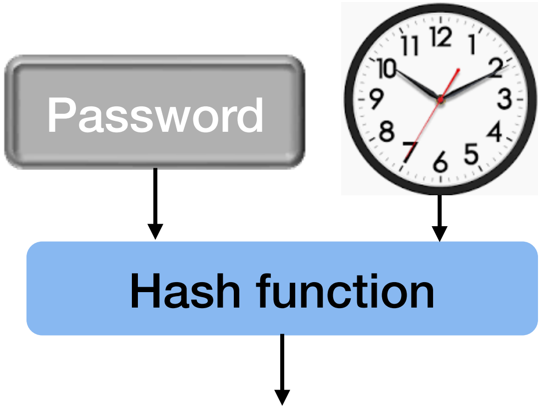
- **Both sides share a secret key**
  - Sometimes sent via a QR code so the user can scan it into the TOTP app
- **User runs TOTP function to generate a one-time password**

  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**



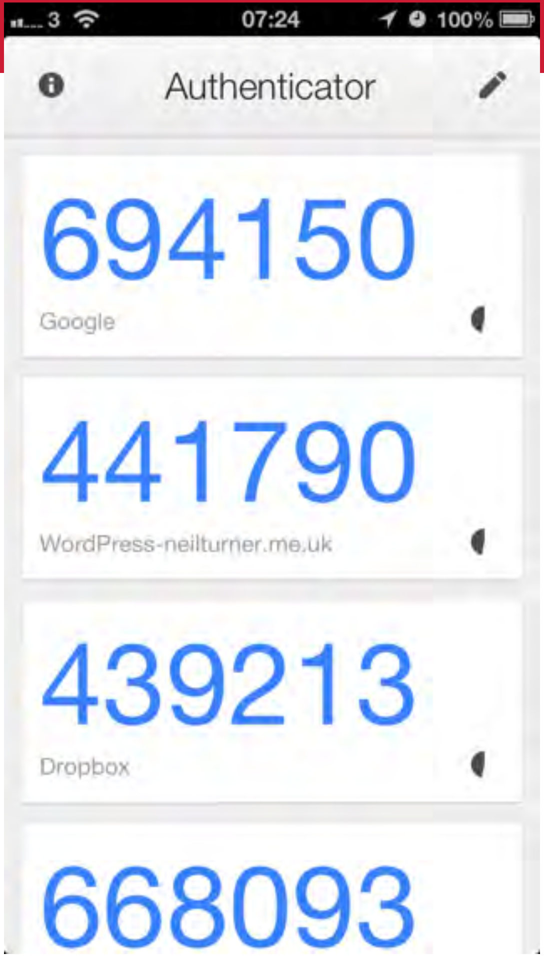
Time-based One-time Passwords

Popular authenticators:

- Microsoft Two-step Verification
- Google Authenticator
- Facebook Code Generator
- Okta
- Duo

Used by

- Microsoft Azure, 365
- Amazon Web Services
- Bitbucket
- Dropbox
- Evernote
- Zoho
- Wordpress
- 1Password
- Many others…

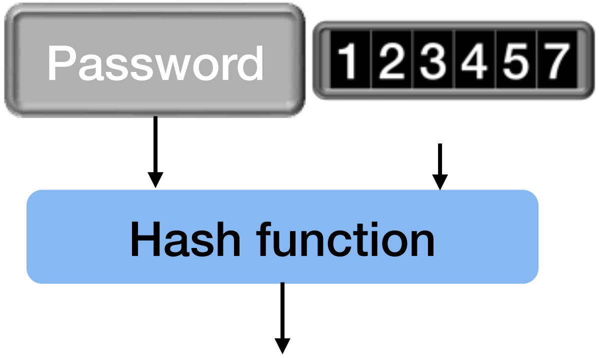


HOTP: Hash-Based One-Time Passwords

- Both sides share a secret key, like TOTP
- Both sides have a counter
- User runs TOTP function to generate a one-time password

  one_time_password = *hash*(secret_key, counter)

- User logs in with: *name*, *password*, and *one_time_password*

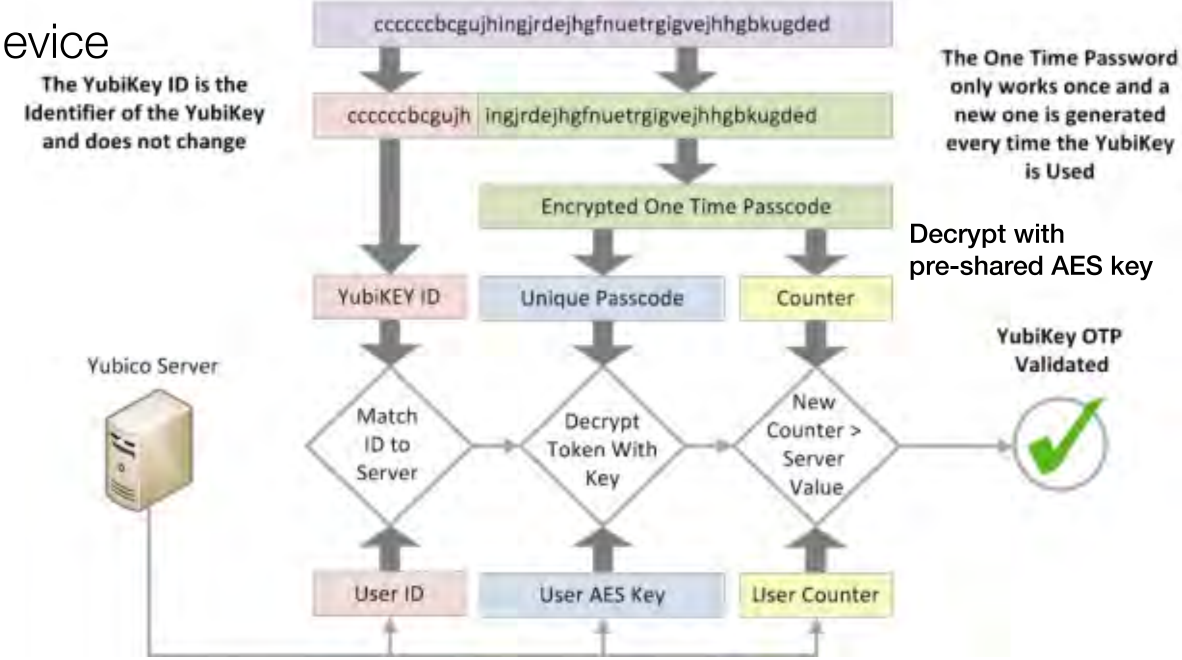


Example Yubikey's Yubico One Time Password

HOTP = Hash-based One-Time Password

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

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



See https://developers.yubico.com/OTP/OTPs_Explained.html

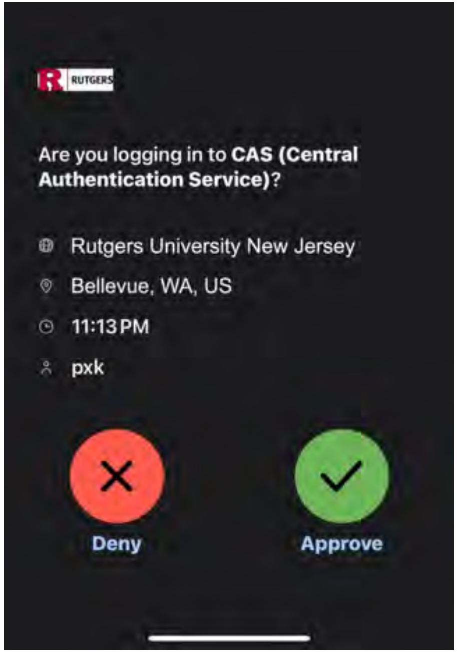
SMS/Email/Push-based Authentication

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

https://www.engadget.com/canada-cryptocurrency-arrest-171617452.html

Number Matching Authentication

- **Push notifications work but may be vulnerable to user fatigue**
  - A careless user might accidentally press *Approve* even if they didn't initiate a login



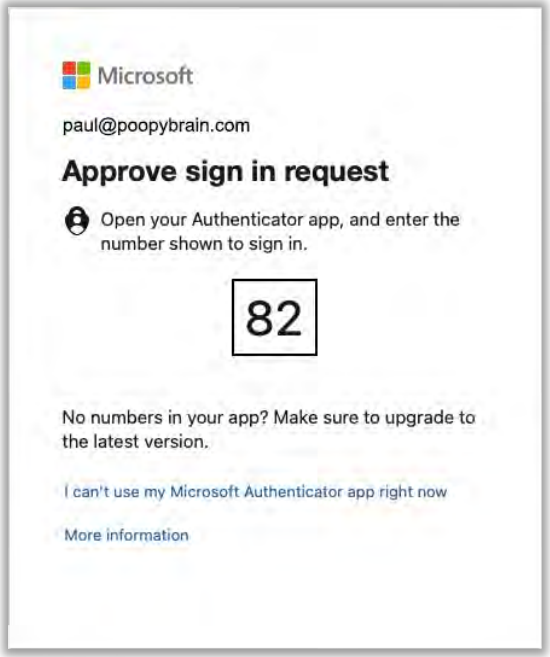
- **Number Matching Authentication forces the user to enter numbers on the authenticator's screen**
  - A login attempt causes the authentication system to:
    - Display a number on the login screen
    - Send a push notification to the user's phone
  - The user has to enter the number they see on the login screen
  - The number is sent to the authentication service
  - If it matches the generated number then the authentication is complete

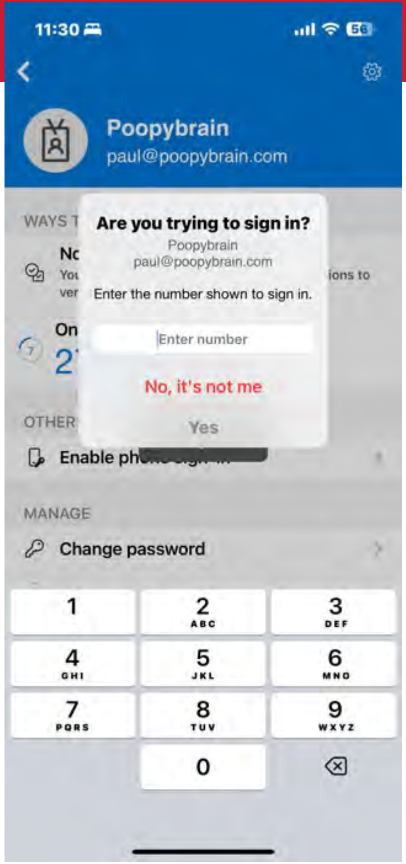
https://www.cisa.gov/sites/default/files/publications/fact-sheet-implement-number-matching-in-mfa-applications-508c.pdf

Number Matching Authentication

Supported by

- Microsoft
- Duo
- Okta





https://www.cisa.gov/sites/default/files/publications/fact-sheet-implement-number-matching-in-mfa-applications-508c.pdf

Adversary in the Middle Attacks (AitM)

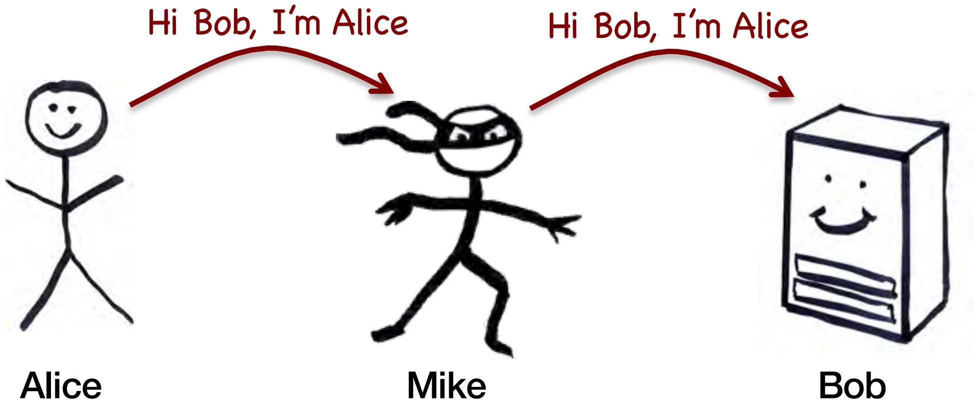
Also known as man-in-the-middle (MitM) attacks

– Attacker acts as the server

Adversary in the Middle Attacks (AitM)

Also known as man-in-the-middle (MitM) attacks

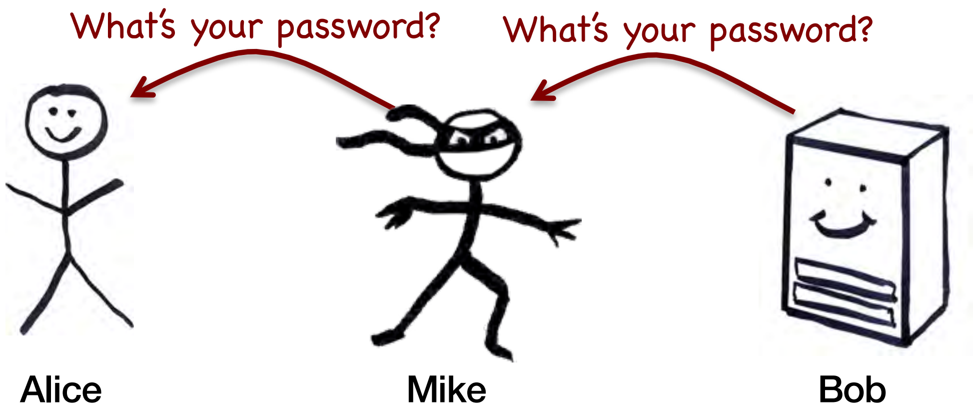
– Attacker acts as the server



Adversary in the Middle Attacks (AitM)

Also known as man-in-the-middle (MitM) attacks

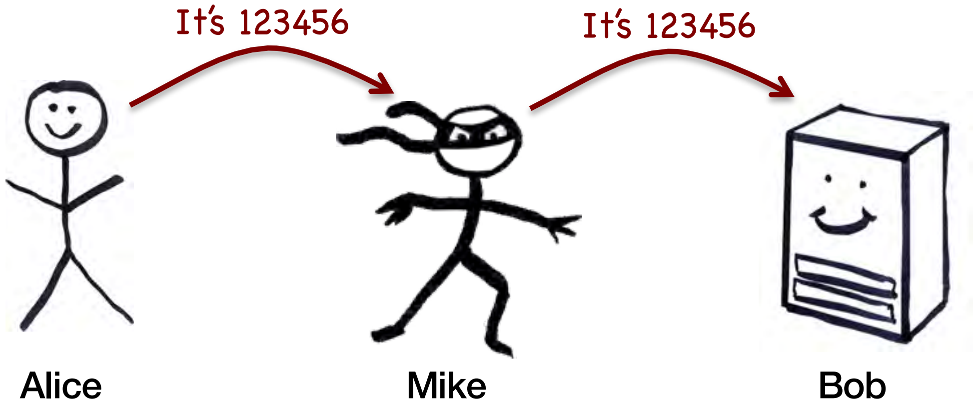
– Attacker acts as the server



Adversary in the Middle Attacks (AitM)

Also known as man-in-the-middle (MitM) attacks

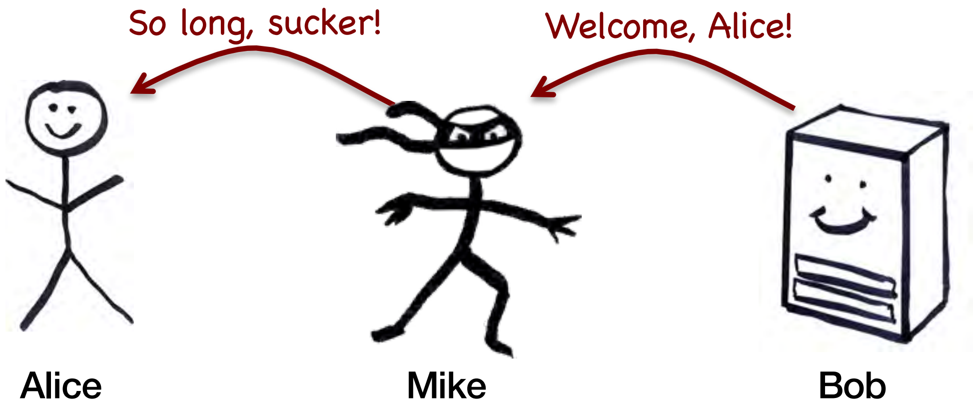
– Attacker acts as the server



Adversary in the Middle Attacks (AitM)

Also known as man-in-the-middle (MitM) attacks

– Attacker acts as the server



Bypassing Two-Factor Authentication (2FA)

- **A cybercrime group known as Sneaky Log has been selling a 2FA-bypassing phishing-as-a-service kit called Sneaky 2FA since late 2023 for $200/month**
- **Adversary-in-the-Middle (AiTM) phishing kit targeting Microsoft 365 accounts**
- **Customers receive obfuscated source code that they can deploy**
  - The code first tries to determine if it's interacting with a human and not a debugger or at a suspicious location (e.g., datacenter)
  - Redirects user to a fake Microsoft authentication page
  - When the authentication is successful, the victim is redirected to a legitimate Office365 error page

https://blog.sekoia.io/sneaky-2fa-exposing-a-new-aitm-phishing-as-a-service/

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

- **Use signed messages for all communication**
  - Signed message = { message, private-key-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