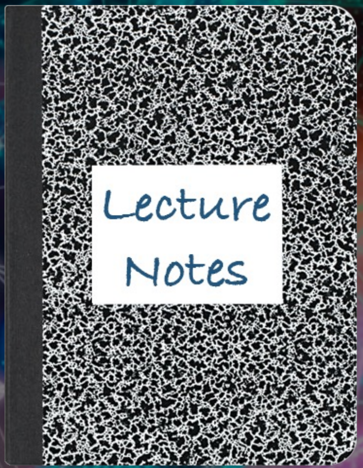


CS 419: Computer Security

Week 8: Authentication

Paul Krzyzanowski

Linux random # generation

Random # generation cleaned up in Linux 5.17 and 5.18

- **/dev/urandom**
  - Would block until there was sufficient entropy detected in the system
  - Danger of excessive (possibly indefinite) blockingg
- **/dev/random & /dev/urandom are now equivalent**
  - Random driver actively adds entropy using the processor's cycle counter (measuring the elapsed time after running the scheduler) if it doesn't have enough

https://www.zx2c4.com/projects/linux-rng-5.17-5.18/

Hundreds of GoDaddy-hosted sites backdoored in a single day

BLEEPINGCOMPUTER

Bill Toulas • March 16, 2022

Internet security analysts have spotted a spike in backdoor infections on WordPress websites hosted on GoDaddy's Managed WordPress service, all featuring an identical backdoor payload. The case affects internet service resellers such as MediaTemple, tsoHost, 123Reg, Domain Factory, Heart Internet, and Host Europe Managed WordPress.

The discovery comes from Wordfence, whose team first observed the malicious activity on March 11, 2022, with 298 websites infected by the backdoor within 24 hours, 281 of which were hosted on GoDaddy.

Old template spammer

The backdoor infecting all sites is a 2015 Google search SEO-poisoning tool implanted on the wp-config.php to fetch spam link templates from the C2 that are used to inject malicious pages into search results.

The campaign uses predominately pharmaceutical spam templates, served to visitors of the compromised websites instead of the actual content.

The goal of these templates is likely to entice the victims to make purchases of fake products, losing money and payment details to the threat actors.

https://www.bleepingcomputer.com/news/security/hundreds-of-godaddy-hosted-sites-backdoored-in-a-single-day/

Weak RSA Public Keys

March 14, 2022 Update

- **Older software generated RSA keys that can be broken instantly with commodity hardware**
- **SafeZone library doesn't randomize the prime numbers well**
  - Used to generate RSA keys
  - After selecting one prime #, the second one is in close proximity to the first
- **Keys generated with primes that are too close together can be broken with Fermat's factorization method, described in 1643**

ars TECHNICA SUBSCRIBE SIGN IN

BREAKING KEYS —

Researcher uses 379-year-old algorithm to crack crypto keys found in the wild

It takes only a second to crack the handful of weak keys. Are there more out there?

DAN GOODIN - 3/14/2022, 5:31 PM

Getty Images

Enlarge

Cryptographic keys generated with older software now owned by technology company Rambus are weak enough to be broken instantly using commodity hardware, a researcher reported on Monday. This revelation is part of an investigation that also uncovered a handful of weak keys in the wild.

The software comes from a basic version of the SafeZone Crypto Libraries, which were developed by a company called Inside Secure and acquired by Rambus as part of its 2019 acquisition of Verimatrix, a Rambus representative said. That version was deprecated prior to the acquisition and is distinct from a FIPS-certified version that the company now sells under the Rambus FIPS Security Toolkit brand.

https://arstechnica.com/information-technology/2022/03/researcher-uses-600-year-old-algorithm-to-crack-crypto-keys-found-in-the-wild/

Weak RSA Public Keys

- **Product of two large primes can be written as**
  N = (*a*-*b*)(*a*+*b*)
  - where *a* is the middle between the two primes
  - *b* is the distance from the middle to each of the primes
- **If the primes are close, then *a* is close to √N**
- **Attack: guess *a* by starting from √N and then incrementing the guess**
  - Calculate $b^2 = a^2 - N$
  - If the result is a square then we guessed correctly
  - Calculate the factors *p*, *q* as $p=a+b$, $q=a-b$

Authentication

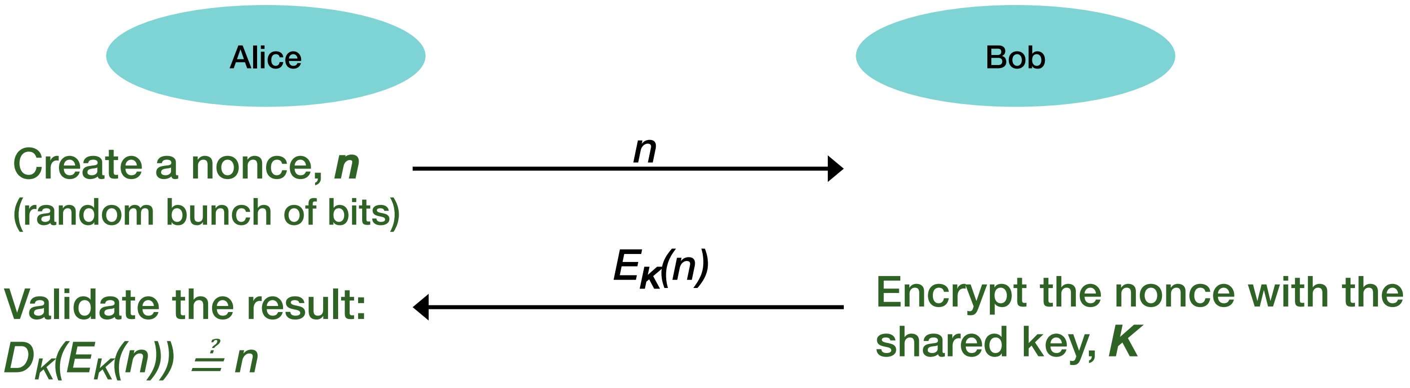
- **Identification**: *who are you?*
- **Authentication**: *prove it*
- **Authorization**: *you can do this*

Some protocols (or services) combine all three

Cryptographic Authentication

The 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

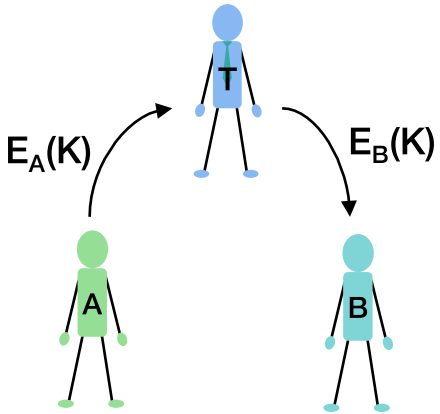
  ⇒ mutual authentication

  – 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 encrypt a message for Trent, she proved she knows her key
  - If Bob can decrypt the message from Trent, he proved he knows his key
- **Trent can also perform *authorization***
- **Weaknesses that we need to address:**
  - Replay attacks

Combined authentication & key exchange algorithms

Security Protocol Notation

$Z \parallel W$

- *Z* concatenated with *W*

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

- *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*

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

- 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

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 → Trent: Request session key to Bob

Alice ← Trent: $\{ k_{A,B} \} k_A \parallel \{ k_{A,B} \} k_B$

Alice → Bob: $\{ k_{A,B} \} k_B$

Alice ↔ Bob: $\{ m \} k_{A,B}$

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

Needham-Schroeder

Add *nonces* – random strings – a

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

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.

$\{ \text{Alice} \parallel \text{Bob} \parallel r_1 \}$

(1) A → Trent

$\{ \text{Alice} \parallel \text{Bob} \parallel r_1 \parallel k_{A,B} \parallel \{ \text{Alice} \parallel k_{A,B} \} k_B \} k_A$

(2) Alice ← Trent

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

$\{ \text{Alice} \parallel k_{A,B} \} k_B$

(3) Alice → Bob

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

$\{ r_2 \} k_{A,B}$

(4) Alice ← Bob

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

(5) Alice → Bob

Needham-Schroeder Protocol Vulnerability

- We assume all keys are secret

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

- But suppose Eve can obtain the session key from an old 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!

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

④ Eve ← Bob: $\{ r_2 \} k_{A,B}$

⑤ Eve → Bob: $\{ r_2 - 1 \} k_{A,B}$

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

Denning-Sacco Solution

- **Problem: replay in the third step of the protocol**
  - Eve 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

Add nonces – random strings – AND a timestamp

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

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

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

Alice ← Bob: $\{ r_2 \} k_{A,B}$

Alice → Bob: $\{ r_2 - 1 \} k_{A,B}$

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

- **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)

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

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

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

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

Bob authenticates himself & forwards request to Trent

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

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

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
- **Supported in most all popular operating systems**
  - Default network authentication used in Microsoft Windows
  - Supported in macOS, Linux, FreeBSD, z/OS, …
  - Used by Rutgers to store 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 | **Kerberos Authentication Service (AS)**

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

→ { "Alice" || "Bob" } →

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

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

ticket

Bob

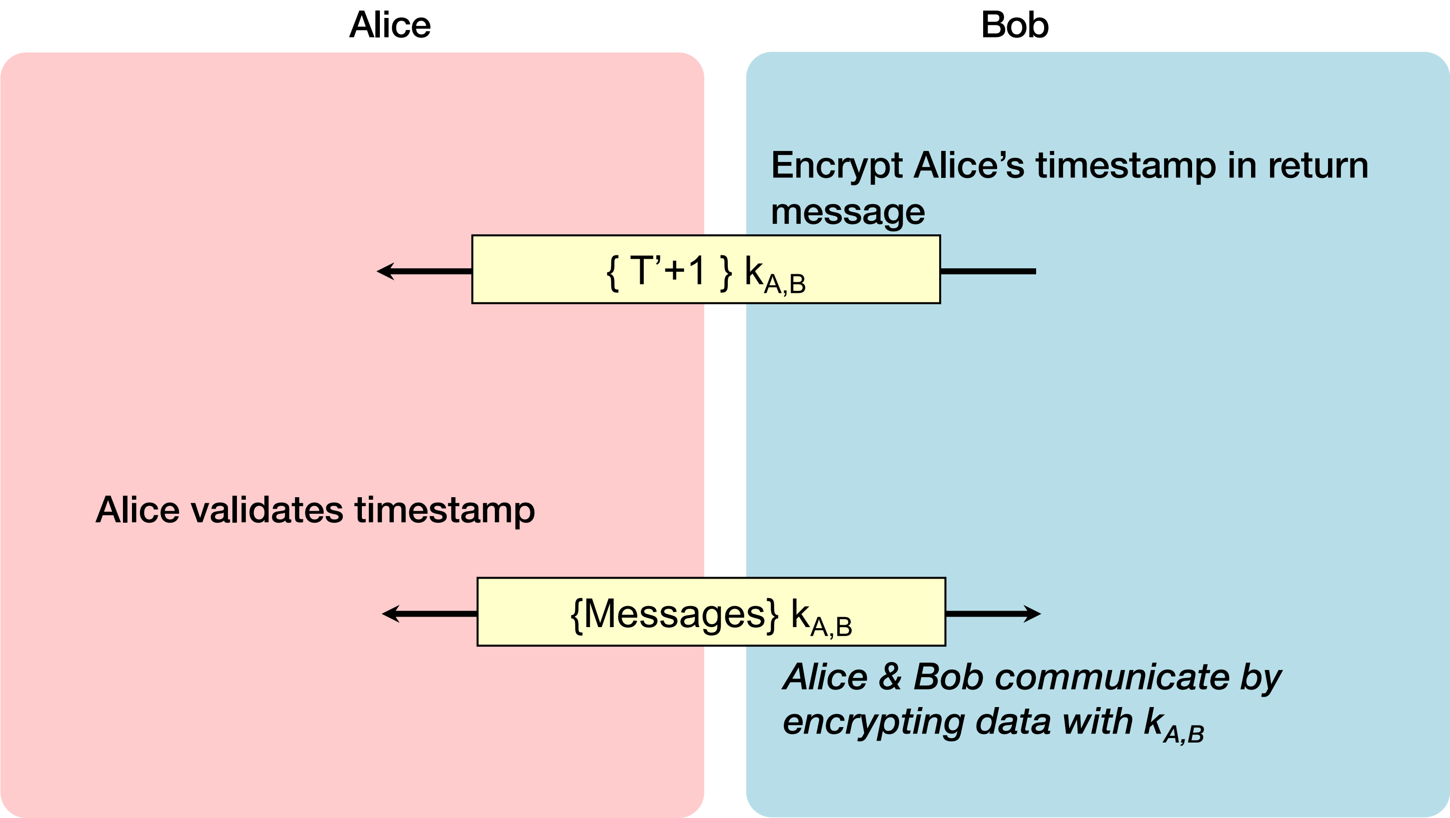
Bob decrypts the ticket:

- 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

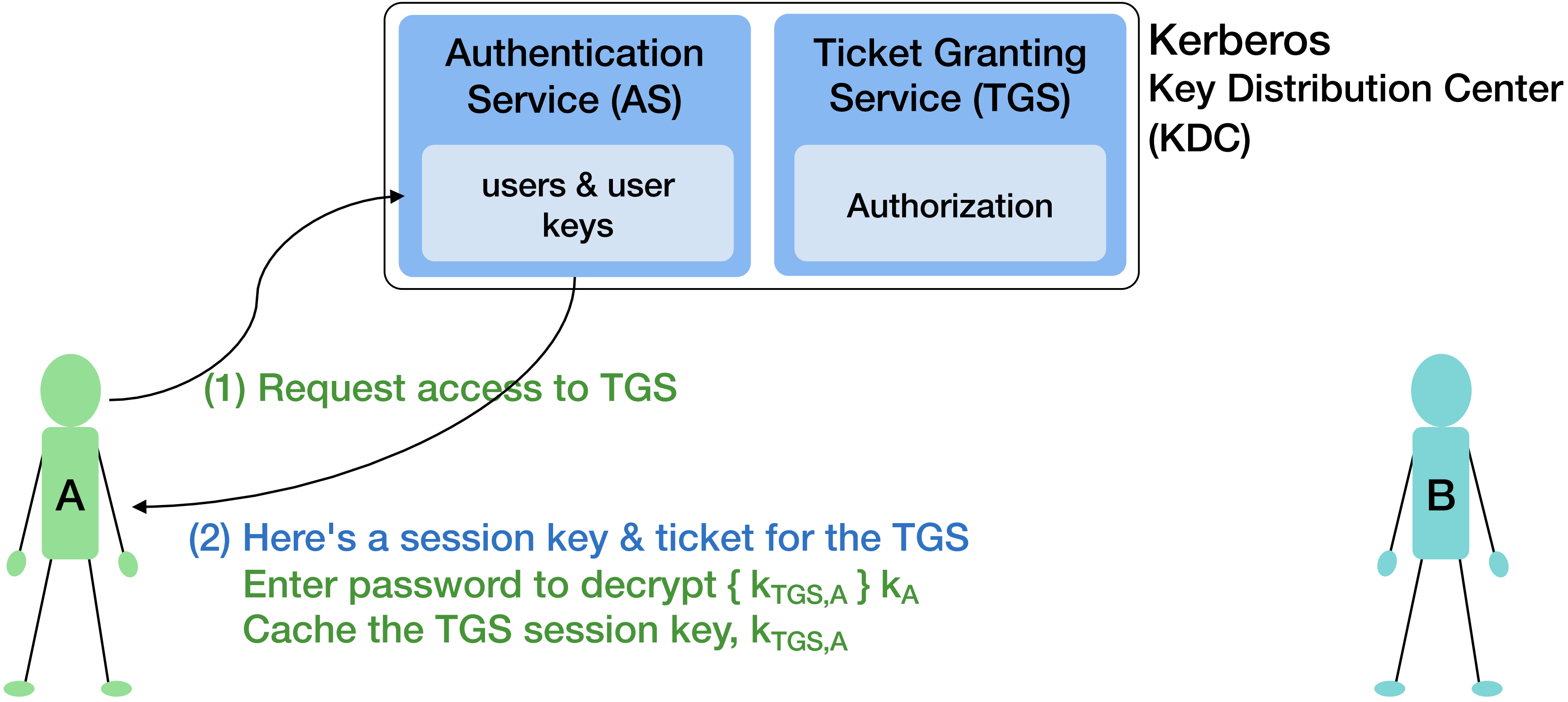
- **Another way: create a temporary password**
  - We can cache this temporary password
  - It's just a session key to access Kerberos – to get access to other services
  - Split Kerberos server into

Authentication Service + Ticket Granting Service

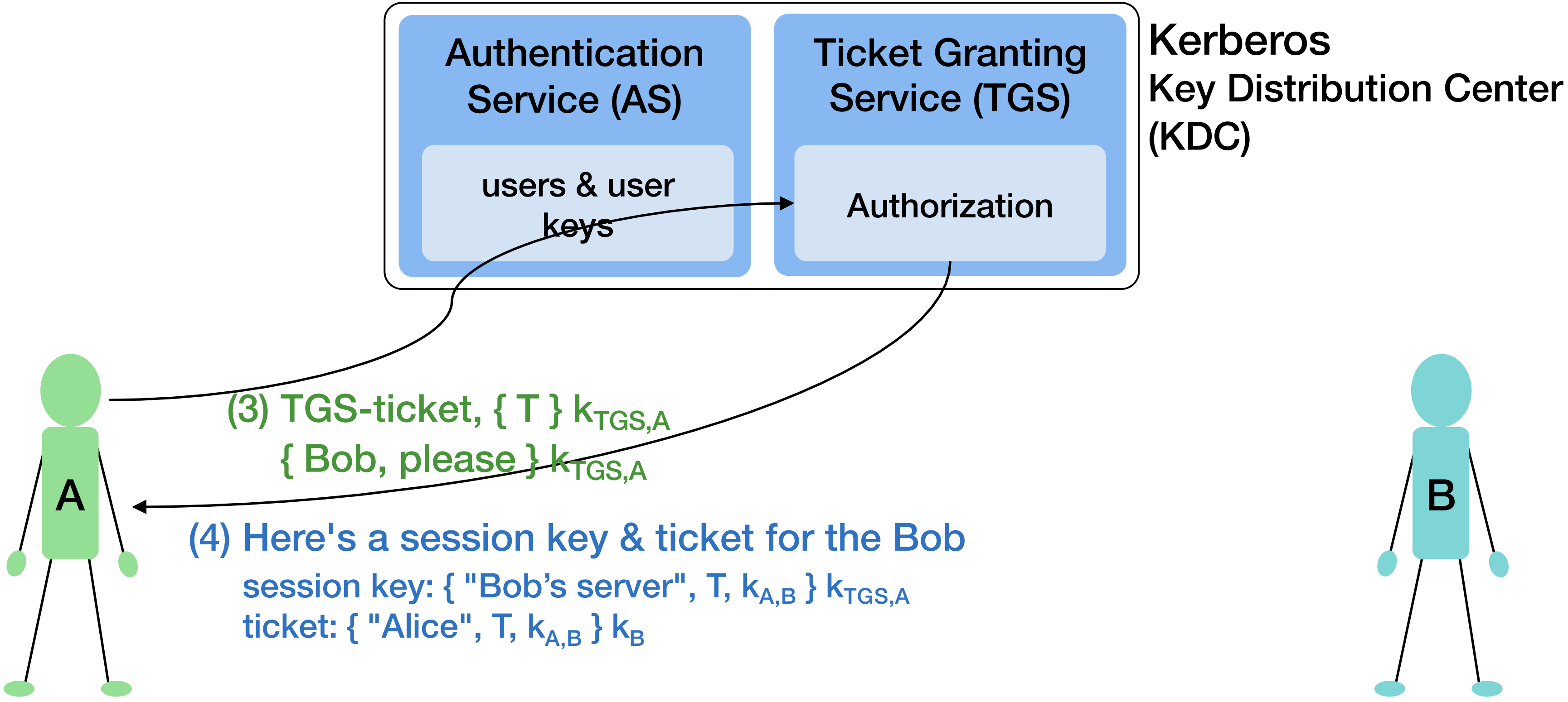
Ticket Granting Server (TGS)

- **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

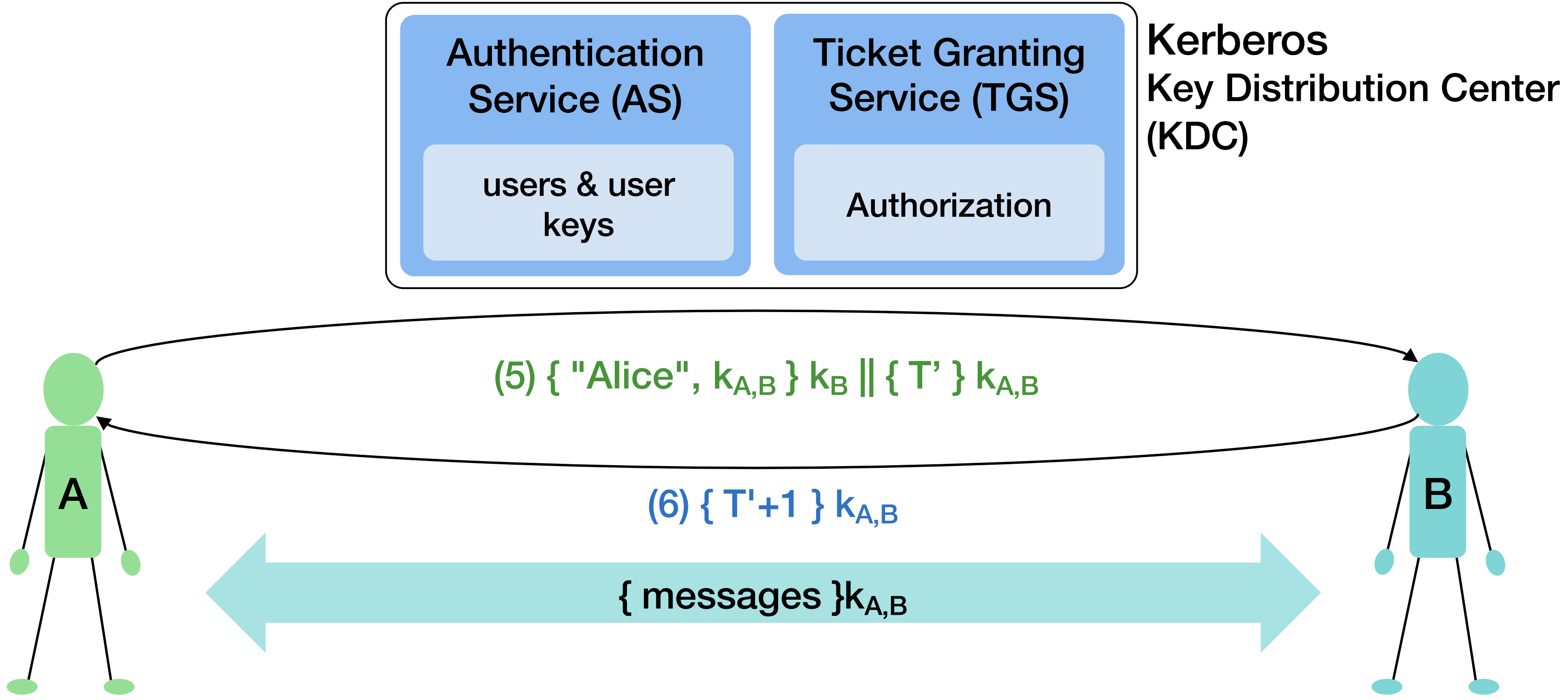
Kerberos AS + TGS: Step 1



Kerberos AS + TGS



Kerberos AS + TGS



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.

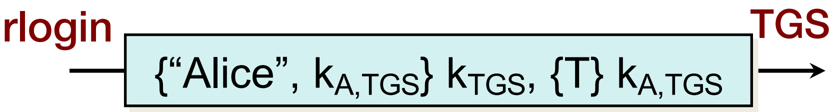
You now have a ticket to access the Ticket Granting Service

Using Kerberos

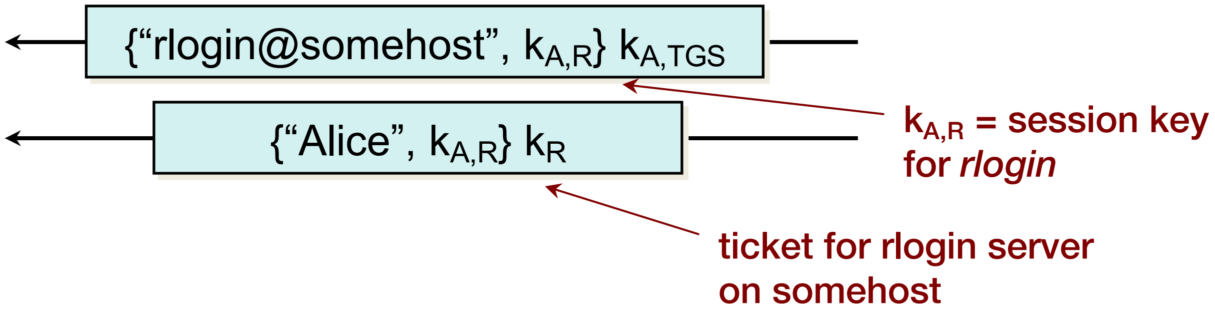
$ rlogin *somehost*

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

Alice sends session key, S, to TGS



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



Combined authentication & key exchange

Basic idea with symmetric cryptography:

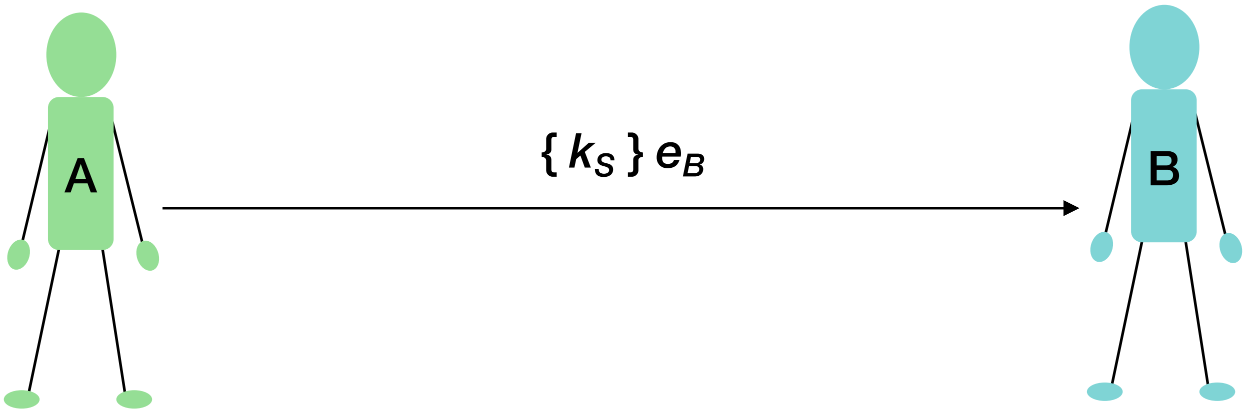
Use a trusted third party (Trent) that has all the keys

- **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

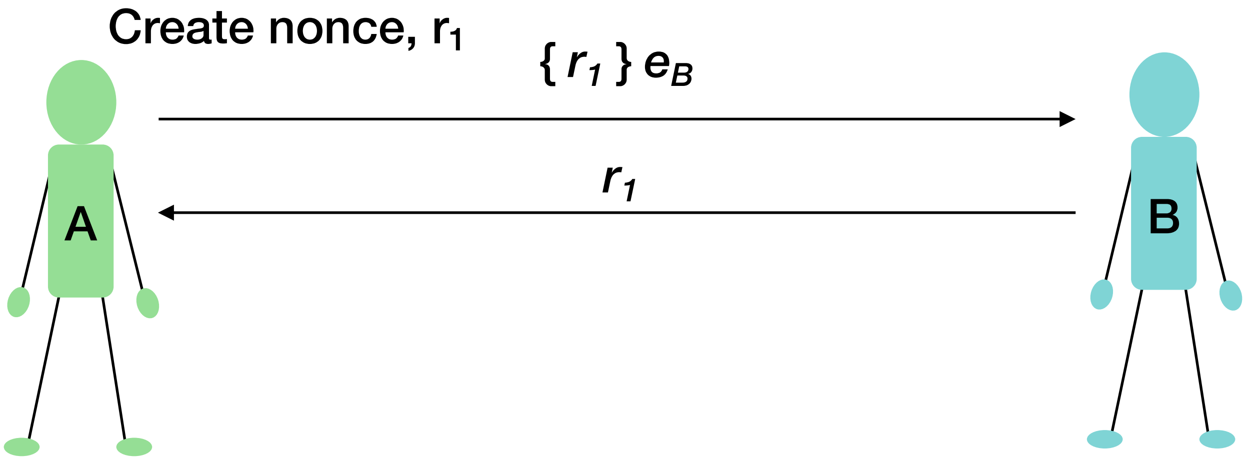
We saw how this works…

- 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:



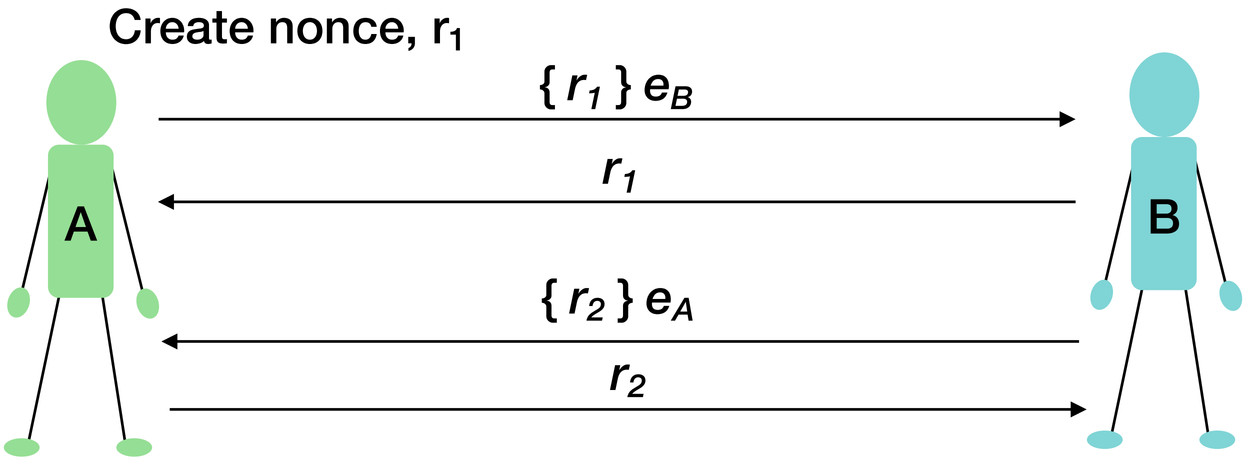
Adding authentication

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



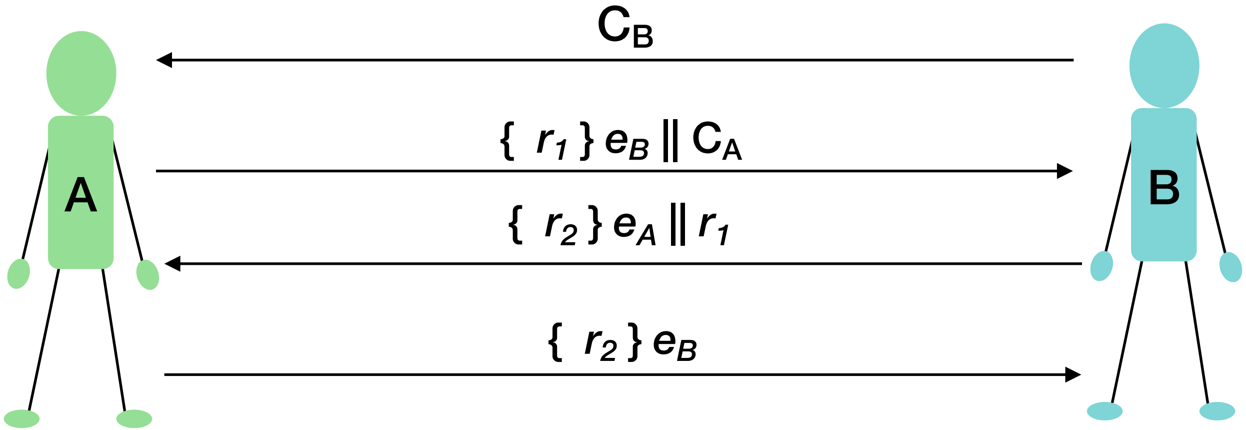
Adding mutual authentication

- 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



Cryptographic toolbox

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

User Authentication

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

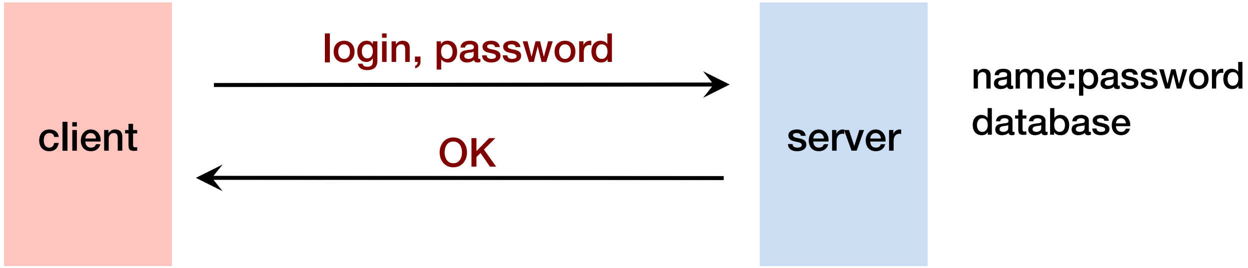
- **Password + code delivered via SMS: 2-factor authentication**
  - Password — something you know
  - Code — something you have: your phone

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	2013	2014	2015	2016	2017	2018	2019	2020	2021
1	123456	123456	123456	123456	123456	123456	123456	123456	123456
2	password	password	password	password	password	password	123456789	123456789	123456789
3	12345678	12345	12345678	12345	12345678	123456789	qwerty	picture1	12345
4	qwerty	12345678	qwerty	12345678	qwerty	12345678	password	password	qwerty
5	abc123	qwerty	12345	football	12345	12345	1234567	12345678	password
6	123456789	123456789	123456789	qwerty	123456789	111111	12345678	111111	12345678
7	111111	1234	football	1234567890	letmein	1234567	12345	123123	111111
8	1234567	baseball	1234	1234567	1234567	sunshine	iloveyou	12345	123123

Top passwords by year 2013-2019: SplashData; 2020-2021: NordPass

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

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

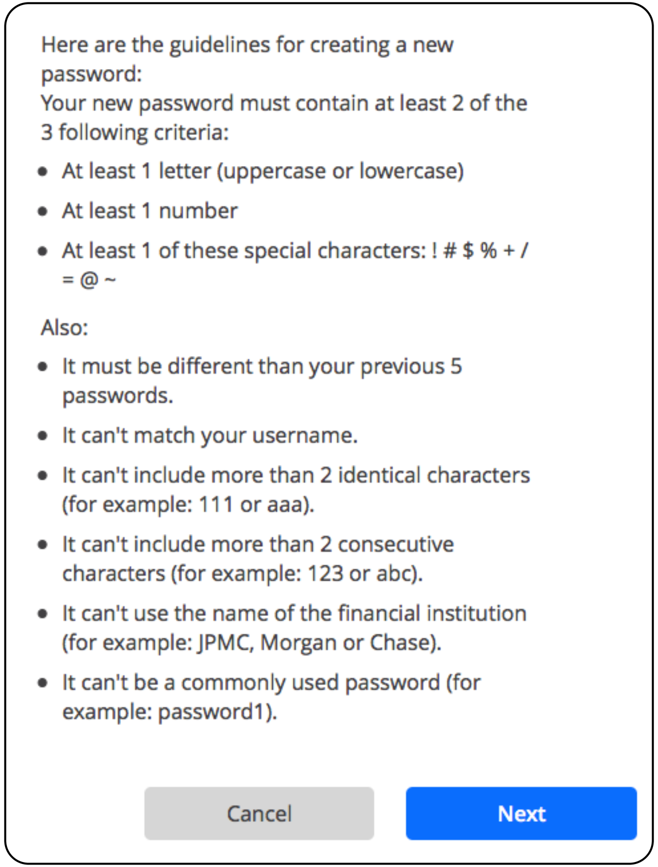
Policies to the rescue?

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



Here are the guidelines for creating a new password:
Your new password must contain at least 2 of the 3 following criteria:

- At least 1 letter (uppercase or lowercase)
- At least 1 number
- At least 1 of these special characters: ! # $ % + / = @ ~

Also:

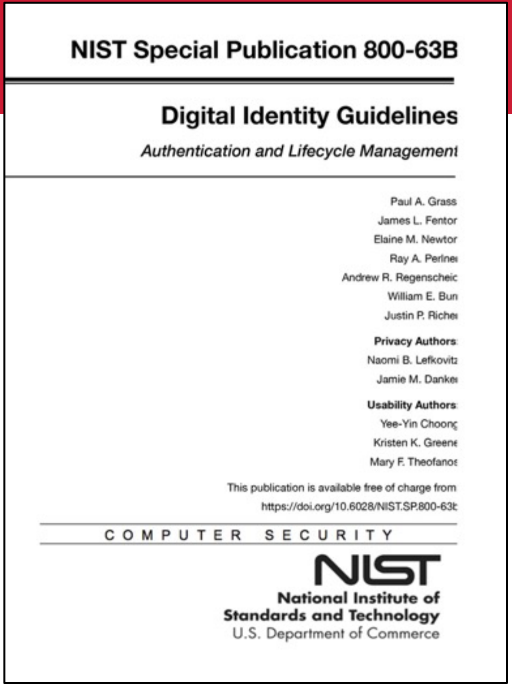
- It must be different than your previous 5 passwords.
- It can't match your username.
- It can't include more than 2 identical characters (for example: 111 or aaa).
- It can't include more than 2 consecutive characters (for example: 123 or abc).
- It can't use the name of the financial institution (for example: JPMC, Morgan or Chase).
- It can't be a commonly used password (for example: password1).

Cancel Next

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. Perlnei
Andrew R. Regenscheic
William E. Bun
Justin P. Riche

Privacy Authors
Naomi B. Lefkovitz
Jamie M. Danke

Usability Authors
Yee-Yin Choonc
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 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

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

Brute force password attacks

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	Instantly	Instantly	Instantly
5	Instantly	Instantly	Instantly	Instantly	Instantly
6	Instantly	Instantly	Instantly	1 sec	5 secs
7	Instantly	Instantly	25 secs	1 min	6 mins
8	Instantly	5 secs	22 mins	1 hour	8 hours
9	Instantly	2 mins	19 hours	3 days	3 weeks
10	Instantly	58 mins	1 month	7 months	5 years
11	2 secs	1 day	5 years	41 years	400 years
12	25 secs	3 weeks	300 years	2k years	34k years
13	4 mins	1 year	16k years	100k years	2m years
14	41 mins	51 years	800k years	9m years	200m years
15	6 hours	1k years	43m years	600m years	15bn years
16	2 days	34k years	2bn years	37bn years	1tn years
17	4 weeks	800k years	100bn years	2tn years	93tn years
18	9 months	23m years	61tn years	100tn years	7qd years

MD5 hashed passwords cracked by a GeForce RTX 2080 GPU

https://www.hivesystems.io/blog/are-your-passwords-in-the-green

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)

"**`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`") =

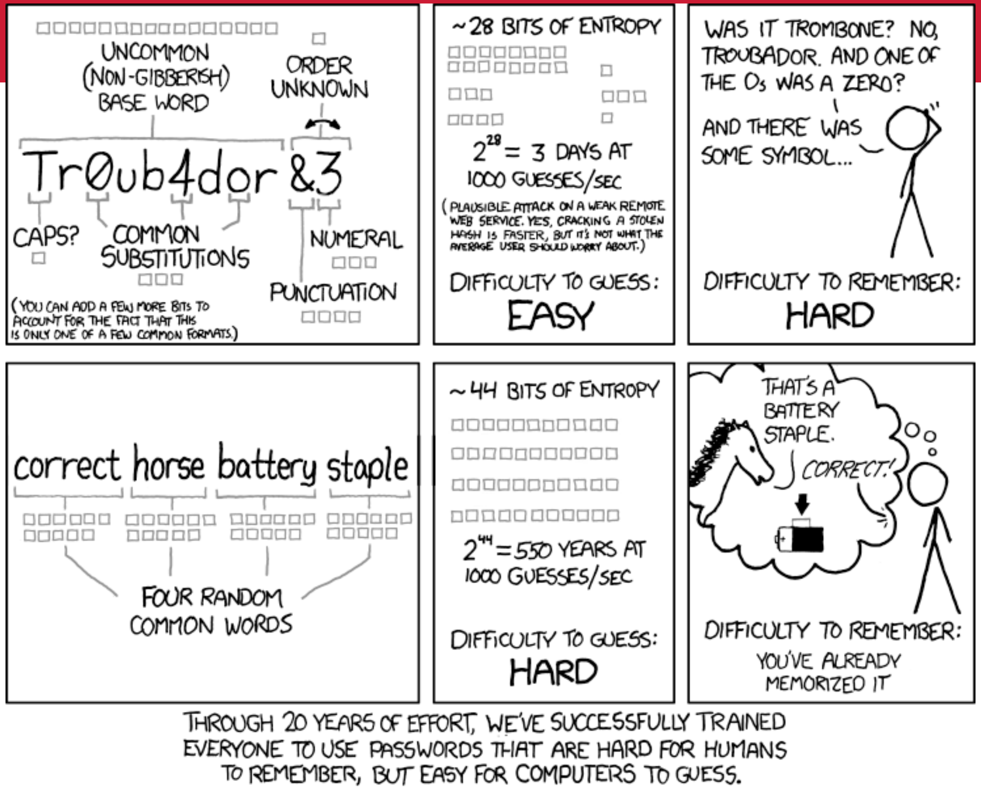
`b791b1b572c0025ef30ecc5fc5ecc5c623f52fca66250560fce8d22623b166c8`

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

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

Defenses

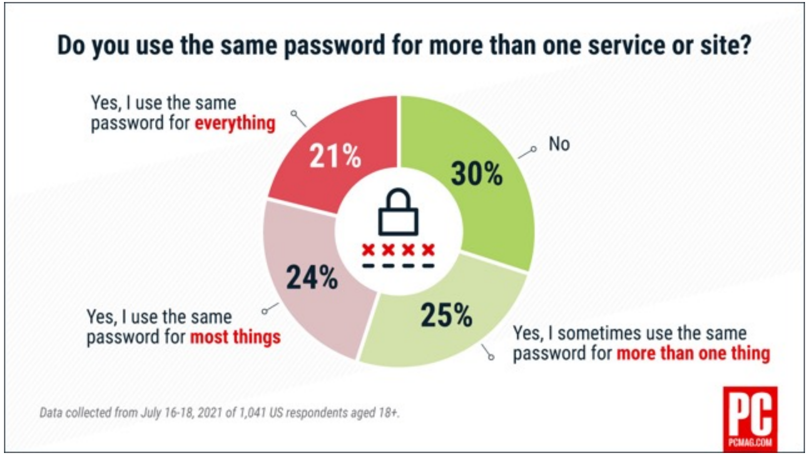
- **Use longer passwords**
  - But can you trust users to pick ones with enough entropy?
- **Rate-limit guesses**
  - Add timeouts after an incorrect password
    - Linux waits about 3 secs – and terminates the *login* program after 5 tries
- **Lock out the account after *N* bad guesses**
  - But this 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 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 username
- **This is the root of phishing attacks**



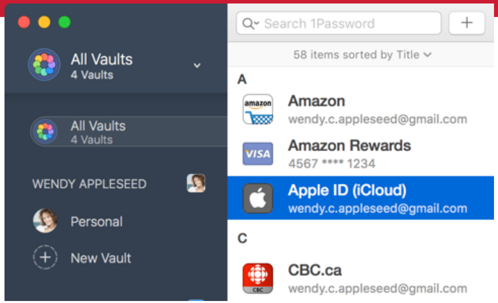
PC Magazine, September 21, 2021

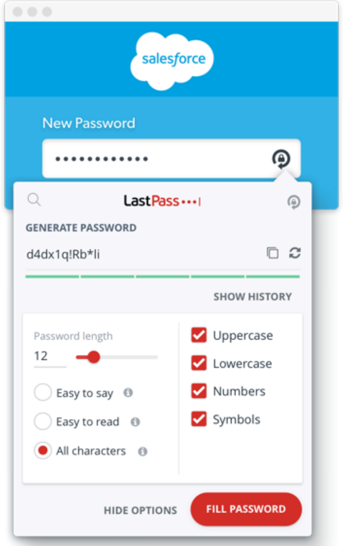
https://www.pcmag.com/news/stop-using-the-same-password-on-multiple-sites-no-really

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 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

THE VERGE

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

Even password managers have security bugs

By Jon Porter • Sep 16, 2019

LastPass has patched a bug that would have allowed a malicious website to extract a previous password entered by the service's browser extension. ZDNet reports that the bug was discovered by Tavis Ormandy, a researcher in Google's Project Zero team, and was disclosed in a bug report dated August 29th. LastPass fixed the issue on September 13th, and deployed the update to all browsers where it should be applied automatically, something LastPass users would be smart to verify.

Key management risks

Password managers are a form of key storage

If attackers get your credentials, they can get all your passwords

South African bank to replace 12m cards after employees stole master key

ZDNet

Postbank says employees printed its master key at one of its data centers and then used it to steal $3.2 million.

Catalin Cimpanu • June 15 2020

Postbank, the banking division of South Africa's Post Office, has lost more than $3.2 million from fraudulent transactions and will now have to replace more than 12 million cards for its customers after employees printed and then stole its master key.

The bank suspects that employees are behind the breach, the news publication said, citing an internal security audit they obtained from a source in the bank.

The master key is a 36-digit code (encryption key) that allows its holder to decrypt the bank's operations and even access and modify banking systems. It is also used to generate keys for customer cards.

https://www.zdnet.com/article/south-african-bank-to-replace-12m-cards-after-employees-stole-master-key/

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**

(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), $f(x)$:

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

Give this list to Alice

- Then compute:

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

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(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}.

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**

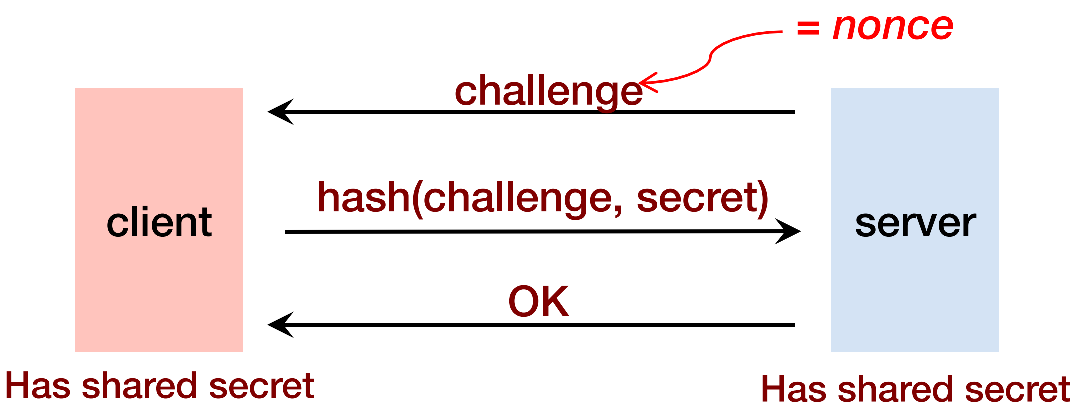
  starting_password = Hash(*seed*, *secret_pass_phrase*)

  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

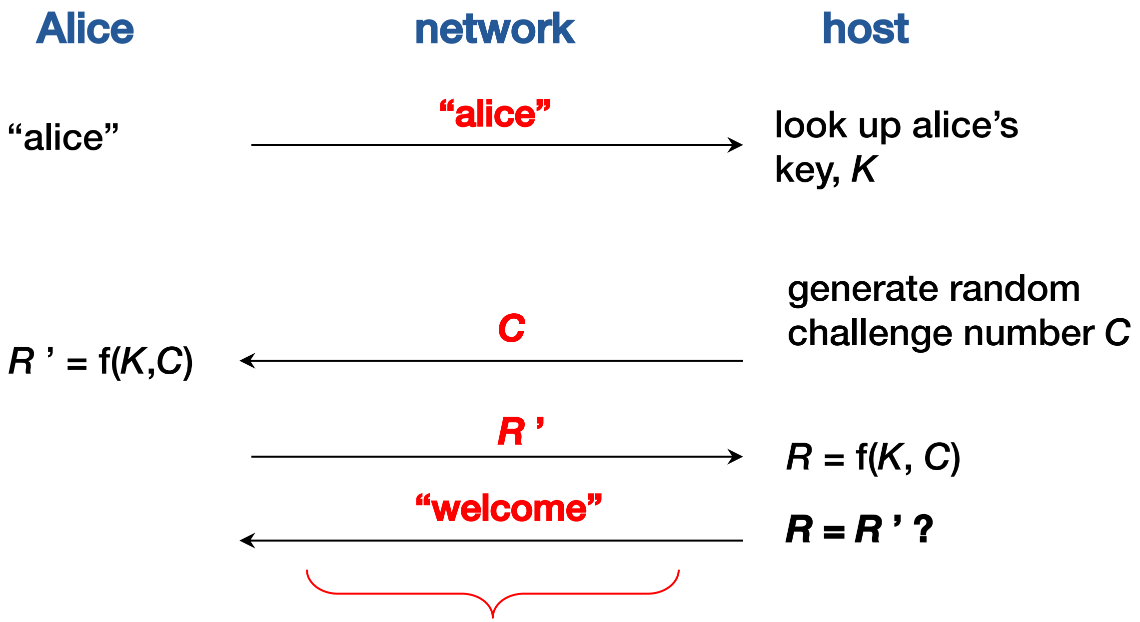


The challenge is a *nonce* (random bits).

We create a hash of the nonce and the secret.

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

CHAP authentication



an eavesdropper does not see K

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

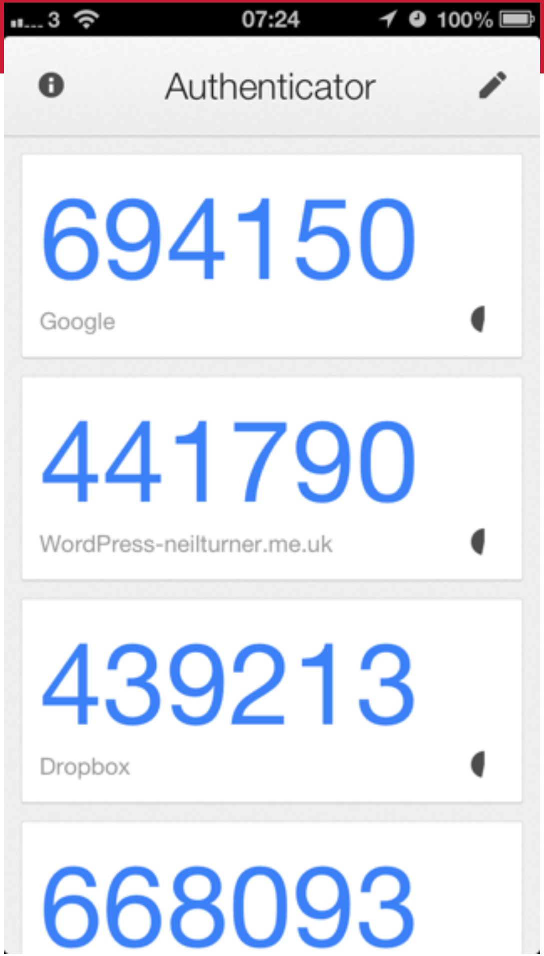
Time-based One-time Passwords

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…



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

SecurID (SASL) authentication: server side

- **Look up user's PIN and seed associated with the token**

- **Get the time of day**
  - Server stores relative accuracy of clock in that SecurID card
  - historic pattern of drift
  - adds or subtracts offset to determine what the clock chip on the SecurID card believes is its current time

- **Passcode is a cryptographic hash of seed, PIN, and time**
  - server computes *f* (seed, PIN, time)

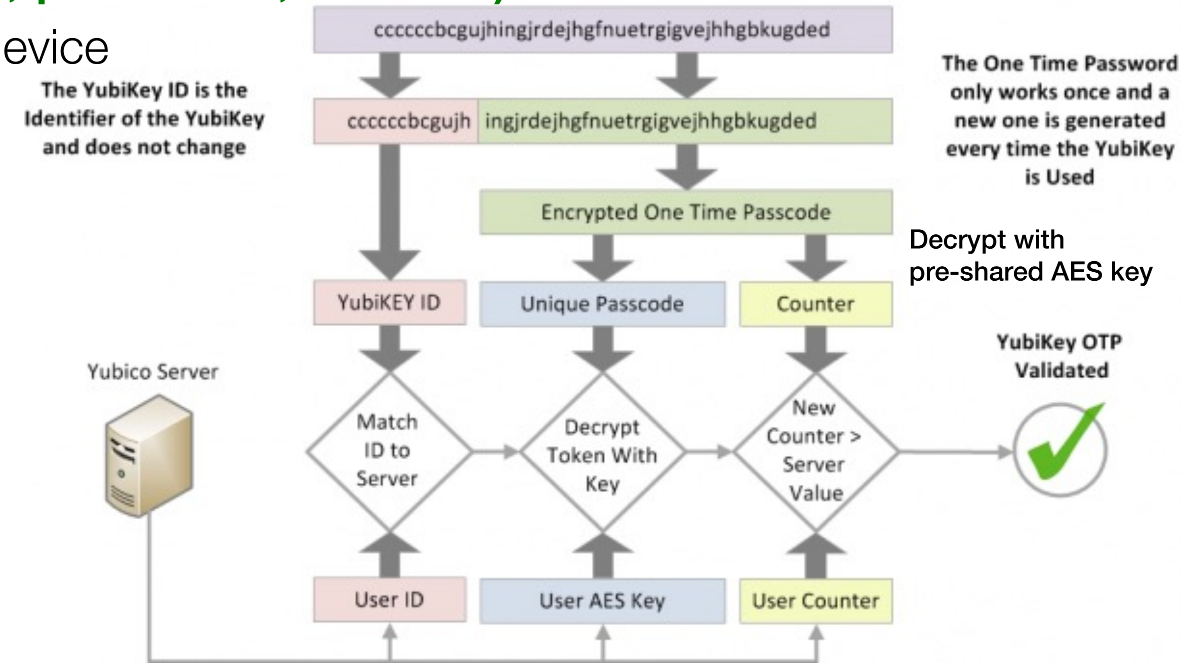
- **Server compares results with data sent by client**

HOTP – Yubikey: 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 Authentication

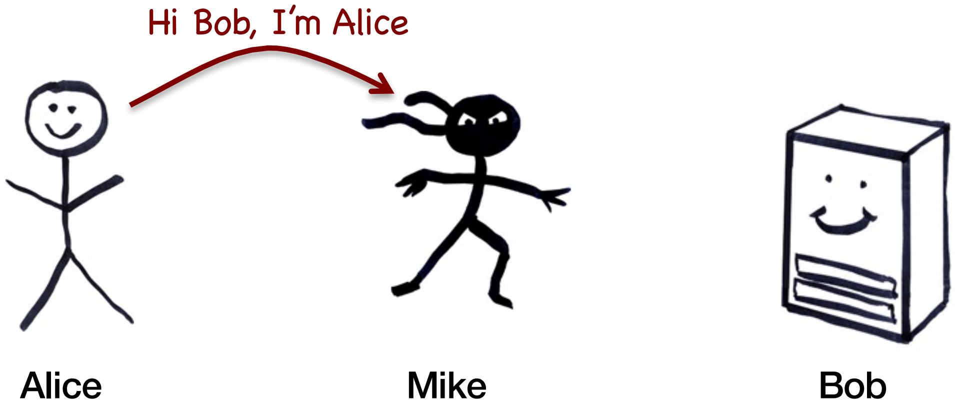
- **Second factor = your possession of a phone (or computer)**
- **After login, sever sends you a code via SMS (or email)**
- **Entering it is proof that you could receive the message**
- **Dangers**
  - SIM swapping attacks
    (social engineering on the phone company)
    - Targeted but viable for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

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

Man-in-the-Middle Attacks (MitM)

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

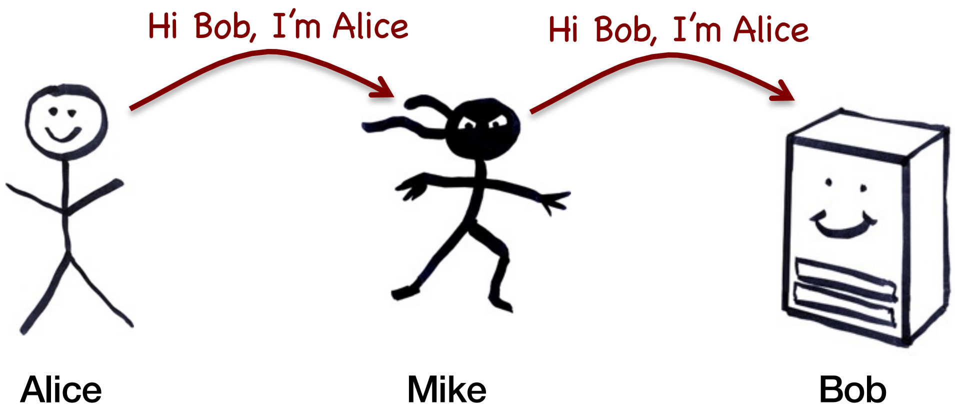
– Attacker acts as the server



Man-in-the-Middle Attacks (MitM)

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

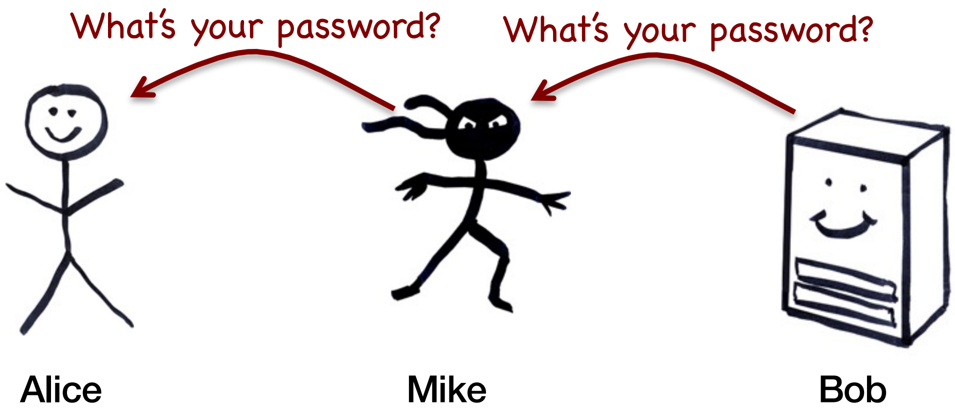
– Attacker acts as the server



Man-in-the-Middle Attacks (MitM)

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

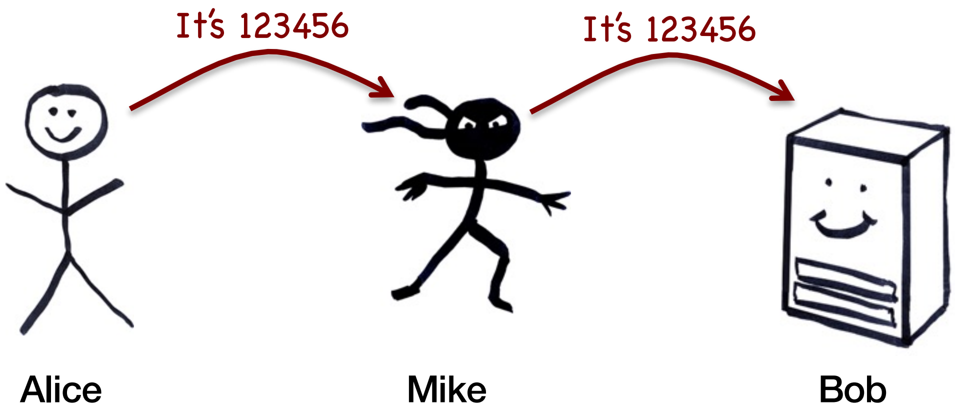
– Attacker acts as the server



Man-in-the-Middle Attacks (MitM)

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

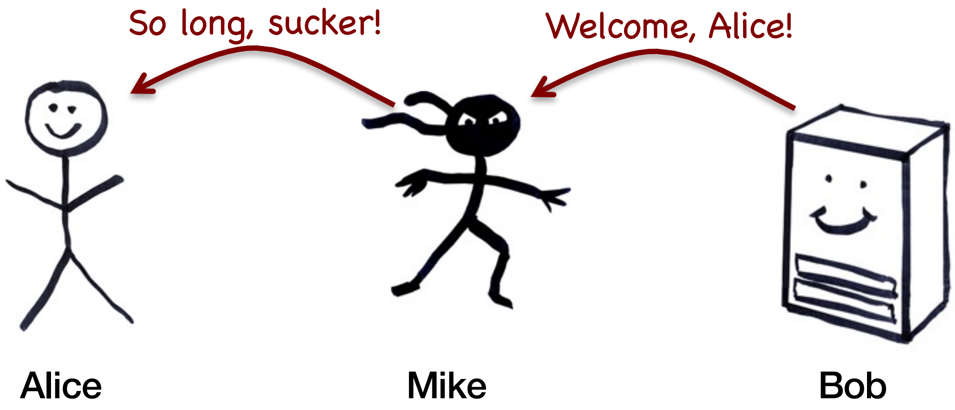
– Attacker acts as the server



Man-in-the-Middle Attacks (MitM)

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

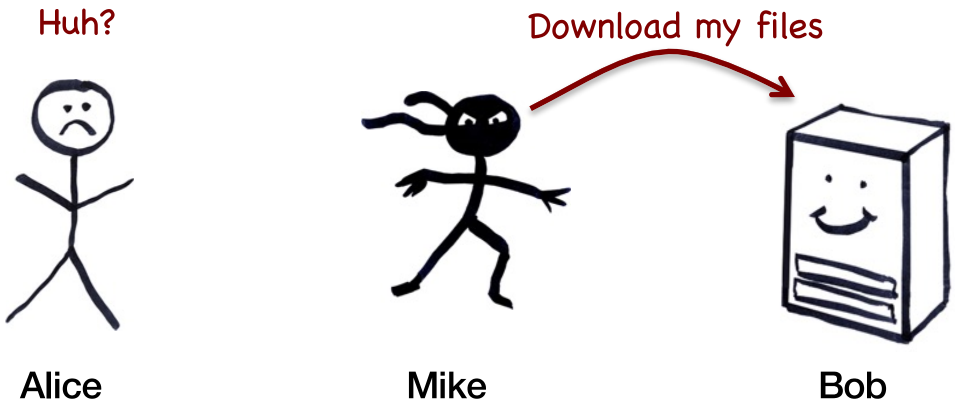
– Attacker acts as the server



Man-in-the-Middle Attacks (MitM)

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
- **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