Computer Security

09. Authentication

Paul Krzyzanowski
Rutgers University
Spring 2019

Authentication

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

- Protocols such as Kerberos combine all three

Authentication

Three factors:

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

Multi-Factor Authentication

Factors may be combined

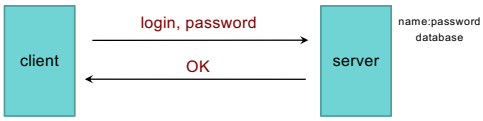
- ATM machine: 2-factor authentication
  - ATM card something you have
  - PIN something you know
- Password + code delivered via SMS: 2-factor authentication
  - Password something you know
  - Code validates that you possess your phone

Two passwords ≠ Two-factor authentication

The factors must be different

Authentication: PAP

Password Authentication Protocol



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

Passwords are bad

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

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

Common Passwords

Adobe security breach (November 2013)
- 152 million Adobe customer records … with encrypted passwords
- Adobe encrypted passwords with a symmetric key algorithm
- … and used the same key to encrypt every password!

Top 26 Adobe Passwords

	Frequency	Password		Frequency	Password
1	1,911,938	123456	14	61,453	1234
2	446,162	123456789	15	56,744	adobe1
3	345,834	password	16	54,651	macromedia
4	211,659	adobe123	17	48,850	azerty
5	201,580	12345678	18	47,142	iloveyou
6	130,832	qwerty	19	44,281	aaaaaa
7	124,253	1234567	20	43,670	654321
8	113,884	111111	21	43,497	12345
9	83,411	photoshop	22	37,407	666666
10	82,694	123123	23	35,325	sunshine
11	76,910	1234567890	24	34,963	123321
12	76,186	000000	25	33,452	letmein
13	70,791	abc123	26	32,549	monkey

It's not getting better

Leaks have not convinced people to use good passwords

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

Past seven years of top passwords from SplashData's list

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

PAP: Reusable passwords

Problem #1: Open access to the password file

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

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

How about encrypting the passwords?

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

PAP: Reusable passwords

Solution:

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

What is a dictionary attack?

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

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

How to speed up a dictionary attack

Create a table of precomputed hashes

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

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

Salt: defeating dictionary attacks

Salt = random string (typically up to 16 characters)

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

"am$7b22QL" + "password"

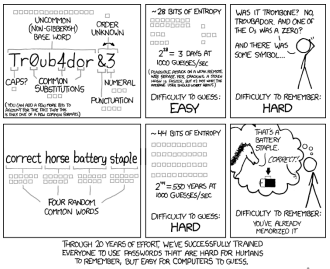
- Even if you know the salt, you cannot use precomputed hashes to search for a password
  (because the salt is prefixed to the password string)

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

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

Longer passwords

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



Assume 95 printable characters

Defenses

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

People forget passwords

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

Reusable passwords in multiple places

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

PAP: Reusable passwords

Problem #2: Network sniffing or shoulder surfing

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

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

Solutions:

(1) Use an encrypted communication channel

(2) Use one-time passwords

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

One-time passwords

Use a different password each time

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

Three forms

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

S/key authentication

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

S/key authentication

Authenticate Alice for 100 logins

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

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

Give this list to Alice

- then compute:
  $x_{101} = f(x_{100}) = f(\ldots f(f(f(R)))\ldots)$

S/key authentication

Authenticate Alice for 100 logins

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

alice: x_{101}

S/key authentication

Alice presents the *last* number on her list:

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

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

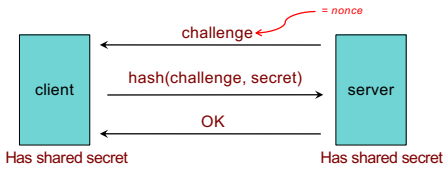
if (x_{100} provided by alice) = passwd("alice")
  replace x_{101} in db with x_{100} provided by alice
  return success
else
  fail

next time: Alice presents x_{99}

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

Authentication: CHAP

Challenge-Handshake Authentication Protocol



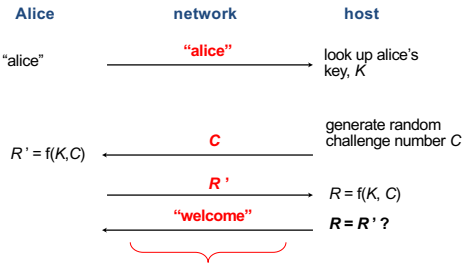
The challenge is a *nonce* (random bits).

We create a hash of the nonce and the secret.

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

April 12, 2019 CS 419 © 2019 Paul Krzyzanowski 25

CHAP authentication



April 12, 2019 CS 419 © 2019 Paul Krzyzanowski 26

Time-Based Authentication

Time-based One-time Password (TOTP) algorithm

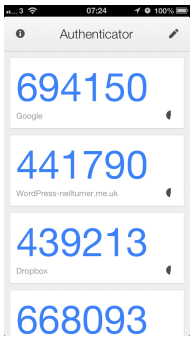
- Both sides share a secret key
  - Sometimes sent via a QR code so the user can scan it into the TOTP app
- User runs TOTP function to generate a one-time password
  one_time_password = *hash*(secret_key, time)
- User logs in with:
  - *Name*, *password*, and *one_time_password*
- Service generates the same password
  one_time_password = *hash*(secret_key, time)
- Typically 30-second granularity for time

April 12, 2019 CS 419 © 2019 Paul Krzyzanowski 28

Time-based One-time Passwords

Used by

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



April 12, 2019 CS 419 © 2019 Paul Krzyzanowski 29

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

April 12, 2019 CS 419 © 2019 Paul Krzyzanowski 30

SecurID card

Same principle as Time-based One-Time Passwords

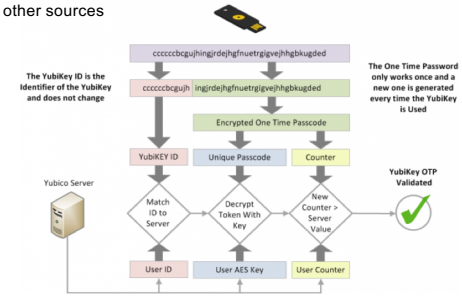
- Proprietary device from RSA
  - SASL mechanism: RFC 2808
- Two-factor authentication based on:
  - **Shared secret key** (seed) ← Something you have
    - stored on authentication card
  - **Shared personal ID** – PIN ← Something you know
    - known by user

April 12, 2019 CS 419 © 2019 Paul Krzyzanowski 31

Yubikey: Yunico One Time Password (OTP)

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

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

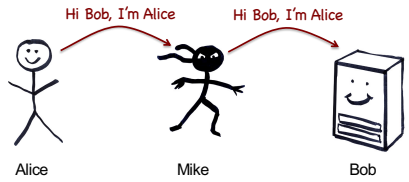


Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server

Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server

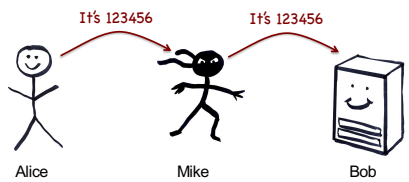


Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server

Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server

Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server

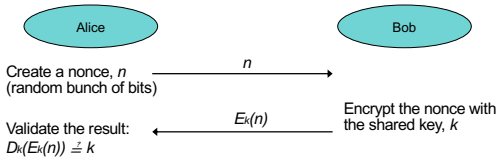
Guarding against man-in-the-middle attacks

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

Cryptographic Authentication

Basic concept: prove you have the key

You can't just ask, so ask the other side to prove they can encrypt or decrypt a message with the key



This assumes a **pre-shared key**.
After that, Alice can encrypt & send a **session key**.

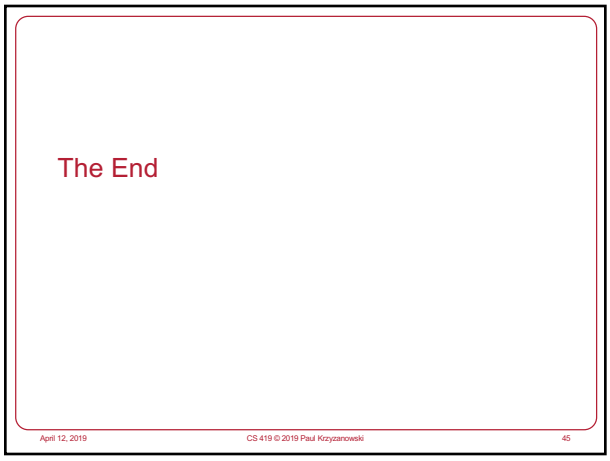
Basic concept: mutual authentication

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

Combined authentication & key exchange

- We looked at this earlier
- 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)


The End
April 12, 2019
CS 419 © 2019 Paul Krzyzanowski
45