User Authentication (ctd.)

System Security (CM0625, CM0631) 2023-24
Università Ca' Foscari Venezia

Riccardo Focardi
www.unive.it/data/persone/5590470
secgroup.dais.unive.it

Università Ca'Foscari Venezia

Preventing leakage and guess

Problem 1: What if the password is *sniffed*?

Solution: only use password over encrypted channels

Example 1: passwords and card numbers sent over **`https`**

Example 2: `telnet` was an **insecure** remote terminal client sending passwords in the clear

Problem 2: What if password is *guessed*?

Solution 1: Disable the service after MAX attempts

Example: lock SIM after 3 attempts

Solution 2: Use strong passwords

⇒ useful in offline attacks when the service cannot be disabled

“Encrypted” passwords

Problem 3: How are password **stored** on the server?

IDEA: The server stores a *one-way hash* of passwords

Definition (***hash function***). A hash function h computes efficiently a **fixed length** value h(x)=z called **digest**, from an x of **arbitrary size**.

Definition (***one-way hash function***). A hash function h is **one-way** if given a digest z, it is ***infeasible* to compute a preimage** x’ such that h(x’)=z

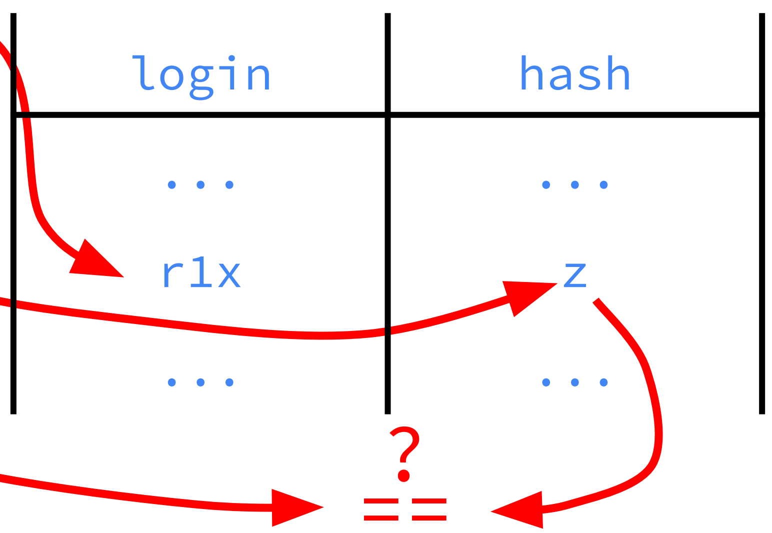
⇒ **Finding** a pre-image is computationally infeasible

Verification of hashed passwords

User is asked for **`login`**, **`pwd`**

The system retrieves the stored hash **`z`** of the password for the given login

The system computes **`h(pwd)`** and checks it is the same as z



⇒ Since h is one-way, in principle, **no password can be recovered from its hash z**

Dictionary attacks

Brute force: even if one-way hashes cannot be inverted, an attacker can try to compute hashes of *easy passwords* and see if the hashes match

Note: It is possible to **precompute** the hashes of a dictionary and just search for z into it

Example:

```
$ echo -n "mypassword" | sha256sum
89e01536ac207279409d4de1e5253e01f4a
1769e696db0d6062ca9b8f56767c8  -
```

Password `"mypassword"` is clearly weak, we can search for the hash directly in search engines or using existing [online services]

Salting passwords

Precomputation of password hashes is prevented by adding a *random salt*

login	hash	salt
...	...	...
r1x	z	s
...	...	...

$$h(pwd,s) \overset{?}{==} z$$

“Slow” hashes

Instead of using a single hash, hashes are usually iterated so to slow down brute-force

Example: Linux passwords

`goofy:$6$Lc5mF7Mm$03IT.AXVhC3Vl4/rLAdomffgv5feOlKBzNGtpEei2dBgK9z/4QBqM3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50: .....`

- `6`: SHA512-based hashing, iterated **5000** times, by default
- `Lc5mF7Mm`: salt
- `03IT.AXVhC3...Zx0Am/y50`: digest

Rainbow tables

Suppose we want to precompute hashes for a huge set of passwords (not just words in a dictionary)

- Storage and searching becomes problematic

Rainbow tables are a technique that allows for a **time/space tradeoff**

- Chains from a password **p** to a final hash **z**
- **p** is hashed and then "reduced" to **p'**
- $\mathbf{p} \rightarrow \mathbf{h(p)} \rightarrow \mathbf{p'} \rightarrow \mathbf{h(p')} \rightarrow \ldots\ \mathbf{p_f} \rightarrow \mathbf{h(p_f) = z}$

Reduction is *any function* returning a candidate pwd

A simple example

```
p = pwd
for (i in [0,C_len-1]):
    print(p)
    h = hash(p)
    print(h)
    p = red(h)
```

`hash` is sha256

`red` takes the first 8 bytes and makes them “printable”

Simple example

donald

4138cfbc5d36f31e8ae09ef4044bb88c0c9c6f289a6a1c27b335a99d1d8dc86f

...

6bI!l%"d

c880c7f068e2b4fe6ec76fea6756d8b1ee92b0d96d0b867be3b952a3ac75cf96

k#j6h(WD

75532eec682a5c65f5a6f8717afc00f67f2518f8bd251865374447cb6bc50725

uS.2h*\e

9d384a0c159b257534258b255023062cbf560491de12ca79ddffca052a5b67b5

@8Jir>%u

6b16a5147f320f182d8d55ed5631203cede6fde5292ba3bd697cb430c2102d22

ksHq"2lu

25f94e180a5abcf4c4c70ab68fc2c6365dee0778e86652fdef8ddeab60d939d2

Searching rainbow tables

Suppose we have **n chains** of **length `C_len`**

(p_1, h_1) (p_2, h_2) … (p_n, h_n)

and we want to invert h

We proceed as follows:

```
r=h, i=0
while (r not in {h1,h2, …, hn} and i < C_len):
    r = hash(red(r))
    i++
```

If h is in the chain we find it!

donald
4138cfbc5d36f31e8ae09ef4044bb88c0c9c6f289a6a1c27b335a99d1d8dc86f
...
6bI!l%"d
c880c7f068e2b4fe6ec76fea6756d8b1ee92b0d96d0b867be3b952a3ac75cf96
k#j6h(WD
75532eec682a5c65f5a6f8717afc00f67f2518f8bd251865374447cb6bc50725
uS.2h*\e
9d384a0c159b257534258b255023062cbf560491de12ca79ddffca052a5b67b5
@8Jir>%u
6b16a5147f320f182d8d55ed5631203cede6fde5292ba3bd697cb430c2102d22
ksHq"2lu
25f94e180a5abcf4c4c70ab68fc2c6365dee0778e86652fdef8ddeab60d939d2

Inverting the hash

If we find the hash after k steps we do

```
r = p  // the password of the matching chain
for C_len - k - 1 steps:
    r = red(hash(r))
return r
```

Inverting the hash

```
adonald
4138cfbc5d36f31e8ae09ef4044bb88c0c9c6f289a6a1c27b335a99d1d8dc86f
A8r_]69{
b6993563cc9fb06b68bc8766b2b556a179557bfb306daade3f032dcf208e9865
Y<5coBSk
1af94c530693bd80abb1bd9eca143324eb3185fbf559634167ece0aa494fd2a1
w?LSc6`#
e5138aee690f1ec23e4fbee436138c51b955b3438a96be23188a7277f1554530
+p-4il{e
93bfa6db6c82dcc1bdf6c9de7682f236817f2e4b25907f7934b0d8d8c28b3107
6bI!l%"d
c880c7f068e2b4fe6ec76fea6756d8b1ee92b0d96d0b867be3b952a3ac75cf96
k#j6h(WD
75532eec682a5c65f5a6f8717afc00f67f2518f8bd251865374447cb6bc50725
uS.2h*\e
9d384a0c159b257534258b255023062cbf560491de12ca79ddffca052a5b67b5
@8Jir>%u
6b16a5147f320f182d8d55ed5631203cede6fde5292ba3bd697cb430c2102d22
ksHq"2lu
25f94e180a5abcf4c4c70ab68fc2c6365dee0778e86652fdef8ddeab60d939d2
```

C_len - 4 - 1 = 10 - 5 = 5

k = 4

Merging chains and space/time tradeoff

Chains can merge, in this case we **lose coverage:** after two chains merge, next hashes will **overlap**

IDEA: Make **red**$_i$ depend on **step i**

⇒ if two chains merge they will split, unless they merge at the very same step!

This is where the name "**Rainbow**" comes from!

P is the set of passwords that we want to cover (assume no collisions)

⇒ We need about |P| / C_len chains (**space decreases** if we increase the chain length)

⇒ Searching **time** is proportional to C_len^2 (notice that with red$_i$ we cannot reuse `red(hash(r))` from previous steps)

Passwords

(summary)

Class: Something known

Passwords are stored “encrypted”:

- **One-way hash** of password and a **random *salt***
- **Iterated hash**: slow down brute force

Rainbow tables are a particularly efficient time-space tradeoff

- Countermeasure: random salt

Attacks on passwords (1)

Offline attacks: through rainbow tables and dictionaries, e.g. the 32M passwords leaked in the rockyou.com SQLi attack of 2009

Countermeasures:

- Salt, slow hashes, password policies (increase **entropy**)
- Restrict access to password file (mitigation)

Online attacks on single accounts: try easy passwords for one account

Countermeasures:

- Lock the account after MAX attempts (e.g. MAX=5)
- Same countermeasure for SIM, smartphone, bank PINs

Attacks on passwords (2)

Popular passwords: try a popular password on many accounts.

Countermeasures:

- Password policies
- Checking IPs (blacklisting)

Password guessing on a single user: guess an easy password for a specific user, e.g., using specific information about that user (name, age, etc.)

Countermeasures:

- User training
- Password policies

Attacks on passwords (3)

Workstation hijacking: wait until a workstation il left unattended (bypass user authentication)

Countermeasures:

- lock after a period on inactivity
- … same for smartphones

User mistakes: written, shared, and preconfigured passwords plus social engineering

Countermeasures:

- User training
- Use of two-factor authentication

Attacks on passwords (4)

Multiple password use: leaking a password that is reused across accounts is more impactful

Countermeasures:

- User training
- Forbid password reuse when possible (e.g. on devices in the same network)

Interception: password should only be transmitted is a secure way

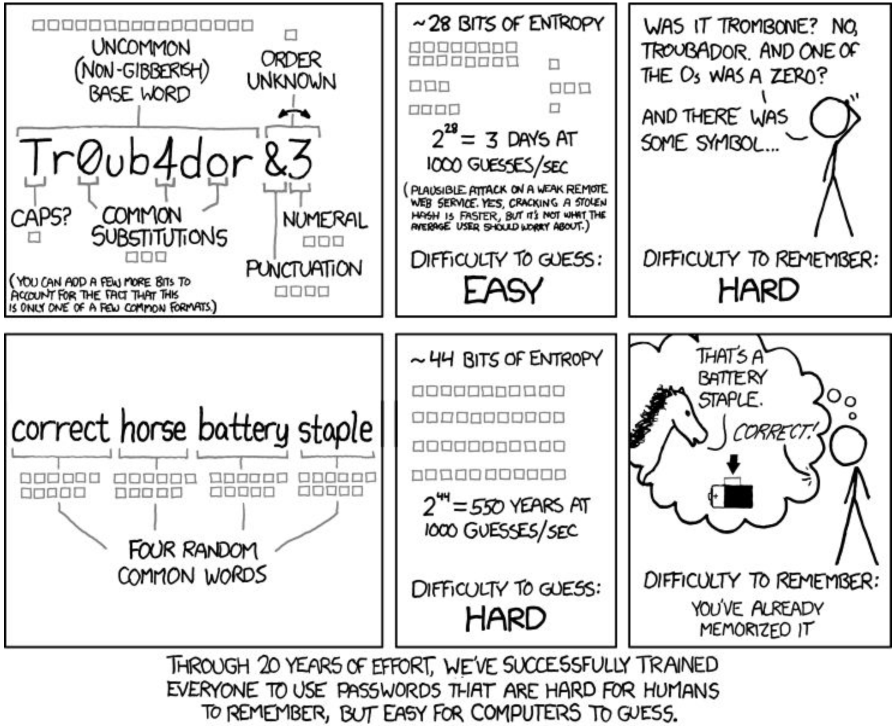
Countermeasures:

- Use a protocol that ensures authenticity of recipients and confidentiality (e.g. TLS)
- …. Just sending the password encrypted does not work!

Password policies

NIST SP 800-63-2 suggests the following alternative rules:

- Password must have at least sixteen characters (**basic16**)
- Password must have at least eight characters including an uppercase and lowercase letter, a symbol, and a digit. It may not contain a dictionary word (**comprehensive8**)



xkcd.com

Diceware

Passphrase of N words picked at random from a fixed list, by rolling 5 dice

- 5 dice gives $6^5 = 7776$ possible words
- Entropy for each word is $\log_2 7776 \sim 12.9$ bits

The **whole entropy** is thus 12.9 * N

- for N=4 entropy is ~**52** bits
- for N=5 entropy is ~**64** bits
- for N=6 entropy is ~**77** bits

Word list: http://world.std.com/~reinhold/dicewarewordlist.pdf

Token-based authentication

Something possessed. Check the **possession** of a device

- ATM cards, credit cards, smartcards, One Time Password (OTP) generators, USB crypto-tokens

Memory cards

Passive card with a memory

Examples:

- Old ATM cards with magnetic stripe
- Hotel cards to open doors

When **paired with a PIN** the attacker needs to steal/duplicate both

Problems:

- Passive cards are usually simple to clone

Example:

- Old ATM cards were cloned by putting a fake reader and a camera (to also steal the PIN)

Smart cards

Smart token with an **embedded chip**

Various devices:

- Standard smartcard
- USB token
- Small portable objects
- Bigger objects with display and/or keyboard

Smart card interface and protocol

Interface:

- **Contact**: a conductive contact plate on the surface of the card (typically gold plated) for transmission of commands, data, and card status
- **Contactless**: Both the reader and the card have an antenna, and communicate using radio frequencies

Protocol:

1. **Static**: token provides a fixed secret (as for passive cards)
2. **One time password** (OTP): the token generates a fresh OTP that is used for authentication
3. **Challenge-response**: a challenge is processed by the token that produces a response (e.g. digitally signed)

One Time Passwords (OTP)

Once a secret is leaked it can be used to authenticate many times:

- sniffed password
- cracked password hash
- cloned passive token

One Time Passwords (OTPs) are <u>never reused</u>

They mitigate password leakage/crack by allowing for a single authentication (es. bank OTPs)

⇒ The token and the computer system must be kept **synchronized** so the computer knows the OTP that is current for this token.

Lamport's hash-based OTP

Given a secret **s** and a **one-way** hash function **h** we compute:

$h^t(s)$ which is: $h(h(\ldots h(s)\ldots))$ **t** times

We let the Claimant and the Verifier share this value

- The Claimant uses the list of passwords:
  $h^{t-1}(s),\ h^{t-2}(s),\ \ldots\ h(s),\ s$
- The Verifier computes $h(pwd)$ and checks if it is equal to the stored hash:
  $h(h^{t-1}(s)) == h^t(s)$
- If the check succeeds the Verifier stores $h^{t-1}(s)$

Lamport's hash-based OTP

passwords:	$h^{t-1}(s)$	$h^{t-2}(s)$	…	$h(s)$	s
stored hashes:	$h^{t}(s)$	$h^{t-1}(s)$	…	$h^{2}(s)$	$h(s)$

Limitation: Only t authentications are possible

Security: Computing next passwords from the current is equivalent to compute the preimage of h, which is **infeasible** (h is one-way)

⇒ More secure than storing a shared secret "seed" used to generate the OTP

Case study: RSA seed breach

RSA SecurID Breach (March 2011)

- The values of secret "seeds" were stored insecurely and have been leaked through phishing

⇒ 40M of devices replaced, big companies attacked, huge image damage for RSA

Biometrics

Something inherent. Check **biometric** features of users

- Signatures, fingerprints, voice, face, hand geometry, retinal patterns, iris, ...

Biometrics

1. **Enrollment**: features are extracted and stored in database
2. **Verification**: features are extracted and compared with the stored ones

A delicate balance:

No impersonation (false positives) but correct users should be identified most of the times (no false negative)

Problem: A breach in the biometric database has **high impact**:

- biometric data is unique, belongs to users
- differently from passwords it cannot be changed if leaked

New attacks: *adversarial machine learning*