Identification

Security 1 2018-19

Università Ca' Foscari Venezia

www.dais.unive.it/~focardi

secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Introduction

Identification is often **required** in order to enforce other security properties

- We identify into a system when we **login**
- We identify to our telephone provider through the sim-card in our **phone**
- We identify to **ATMs** with our cards and PINs

Any time the **access to a resource** needs to be regulated, some form of identification is necessary

Identification == entity authentication

Identification can be though as **authenticating a user** or, more generally, an **entity**

- Allow a **verifier** to check **claimant's** identity

Example: login-password scheme

- The user **claims** her identity by inserting the login
- The system **verifies** the identity by asking for a **secret password**

Properties

An identification scheme should always prevent:

Impersonation, even observing previous identifications

Uncontrolled transferability: the verifier should not **reuse** a previous identification to impersonate the claimant with a different verifier, unless **authorized**

- The verifier has more information available than an attacker, e.g., when the communication is encrypted

Classes of identification schemes

Something known. Check the **knowledge** of a secret

- passwords, passphrases, Personal Identification Numbers (PINs), cryptographic keys

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

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

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

- Paper signatures, fingerprints, voice and face recognition, retinal patterns

Passwords

The identity claimed through the **login** information is checked by asking for a corresponding **secret password**

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

- stolen passwords allows for **impersonation** (*weak authentication*: the secret is exposed)

Problem 2: How are password **stored** on the server?

“Encrypted” passwords

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

Definition (*one-way hash function*). A one-way hash h is a function such that:

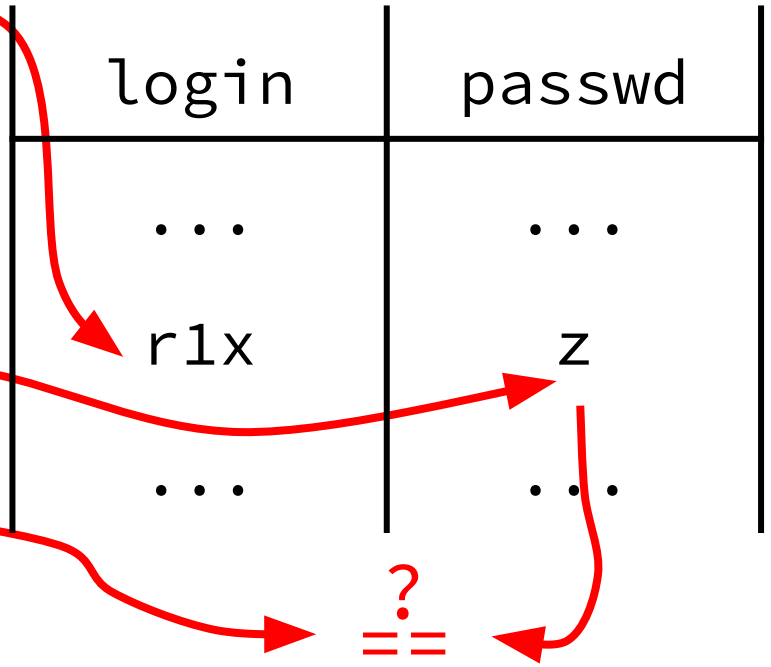
1. given arbitrary data x, the fixed length value h(x)=z called *digest*, can be computed efficiently
2. given a digest z, it is ***infeasible*** **to compute a pre-image** x such that h(x)=z

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 (online service)

Example:

```
r1x@testbed ~ $ echo -n "r1x" | sha256sum
bd269c2940130986b356336f6cbee193c86a7a448d1493ee
feeabdaf6b38a20d  -
```

Salting passwords

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

login	passwd	salt
...	...	...
r1x	z	s
...	...	...

h(pwd,s) ?== 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/rLAdomffgv5feOlKBzNGtp
Eei2dBgK9z/4QBqM3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50: .....
```

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

Example ctd.

Linux passwords in python:

```
>>> import crypt
>>> crypt.crypt("donald","$6$Lc5mF7Mm$")
'$6$Lc5mF7Mm$03IT.AXVhC3Vl4/rLAdomffgv5feOlKBzNGtp
Eei2dBgK9z8B/4QBqM3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50'
```

Command line tool:

```
$ mkpasswd donald -m sha-512 -S Lc5mF7Mm
$6$Lc5mF7Mm$03IT.AXVhC3Vl4/rLAdomffgv5feOlKBzNGtpE
ei2dBgK9z8B/4QBqM3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50
```

Increasing the iterations

```
$ time mkpasswd donald -m sha-512 -S Lc5mF7Mm
...
real    0m0.031s

$ time mkpasswd donald -m sha-512 -S Lc5mF7Mm -R
5000000
$6$rounds=5000000$Lc5mF7Mm$FWm/GeTLTryHa0Nt/WfrbLq
jVOsipSBNP3IUgwbNP7H95eR8lhKj.6Pc7YcznupXjHXA9QBir
kmmaxh3oqt4v.
real    0m22.938s
```

Salt

Up to 16 random chars from [a-zA-Z0-9./]

```
$ mkpasswd donald -m sha-512
$6$KKOpwblUx4TOKQh$I89vJNsqdMWxD2YnbzIjKr2pyPeSUr2b3sFR
AfErzCS8TeX2SqsTECkcKGyzzSNH3i0r8rGHbK3WHfEsR4enI0

$ mkpasswd donald -m sha-512
$6$HT8.lP437G$gNsyr/99EGhvnvNcTlVTdhkf0CcZwoZ90t77Tk7Sq
OBpEDaLZCUy.BD49hXu97CnVZEP8FOg8J9Jr0b09xfn./

$ mkpasswd donald -m sha-512
$6$5GO5Jux.FZyNPiQ$wfmv.u7cfCCaETYkc9ZD93saa91awPfZulJO
AFwYJcr5.AWIxvpqoa.ejns5Gg87MciJm4FNe2Udo.z7rXP4B/
```

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 hased and then "reduced" to **p'**
- $\mathbf{p \rightarrow h(p) \rightarrow p' \rightarrow h(p') \rightarrow \ldots p_f \rightarrow h(p_f) = z}$

Simple example

Reduction is *any function* returning a candidate pwd

```python
#!/usr/bin/python3
import hashlib,binascii
# Reduction: takes the first 8 bytes and makes them writable
def red(z):
    r = ''
    for c in z[:8]:
        r += chr( ( (c-33) % 93 ) + 33)
    return r

p = "donald"
for _ in range(10):
    z = hashlib.sha256(p.encode()).digest()  # computes the digest
    print(p)
    print(binascii.hexlify(z).decode())
    p = red(z)                               # applies reduction
```

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 = red(hash(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 = h
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

Chains can merge, in this case we **lose coverage**

- After two chains merge, next hashes will **overlap**

IDEA: Make **red**$_i$ reduction function **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!

Space/time tradeoff

- 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)

One Time Passwords (OTP)

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

- sniffed
- cracked

One Time Passwords (OTPs) are never reused

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

Lamport's hash-based OTP

Given a secret s and a one-way hash function h we compute: $\mathbf{h^t(s)}$, i.e., $h(h(\ldots\ h(s)\ldots\ ))$ 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)