User Authentication

System Security (CM0625, CM0631) 2025-26
Università Ca' Foscari Venezia

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

Università Ca'Foscari Venezia

Introduction

Identification is the task of correctly identifying a user or entity

It is typically **required** for enforcing other security properties

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

Examples:

- Users identify into a system when they **login**
- Users identify to mobile network providers through the **SIM card**
- Users identify to the SIM card through a **PIN**
- Users identify to **ATMs** with cards and PINs

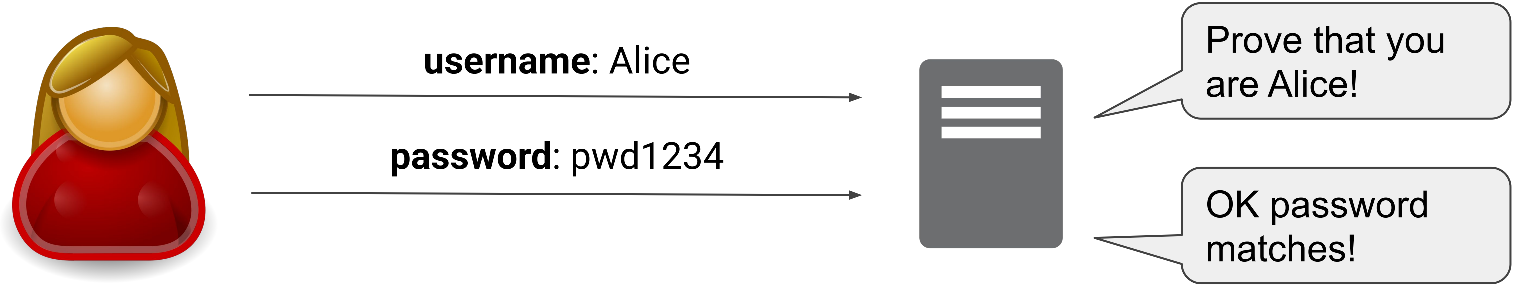
Identification == entity authentication

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 **username**
- The system **verifies** the identity by asking for a **secret password**



Properties

An identification scheme <u>should always prevent</u>:

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
- **Example**: same password for different web sites!

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

Password-based authentication

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

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

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

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
...	...	...

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

“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/rLAdomffgv5feOlKBzNGtpEei
2dBgK9z/4QBqM3ZMRK4qcbbYJhkAE.2KscEZx0Am/y50: .....
```

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

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 **<u>one-way</u>** hash function **h** we compute:

$$\mathbf{h^{t}(s)} \qquad \text{which is: } \mathbf{h(h(\dots\ h(s)\dots\ ))} \quad \mathbf{t} \text{ times}$$

We let the Claimant and the Verifier <u>share this value</u>

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

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 1: RSA seed breach

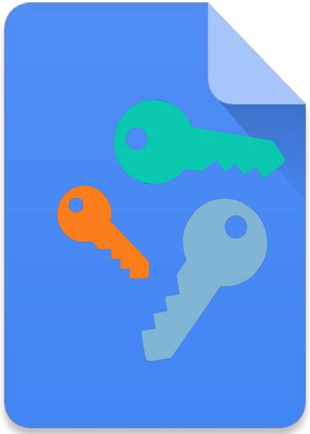
RSA SecurID Breach (March 2011)

- 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

Case study 2: Java keystores

Key Storage



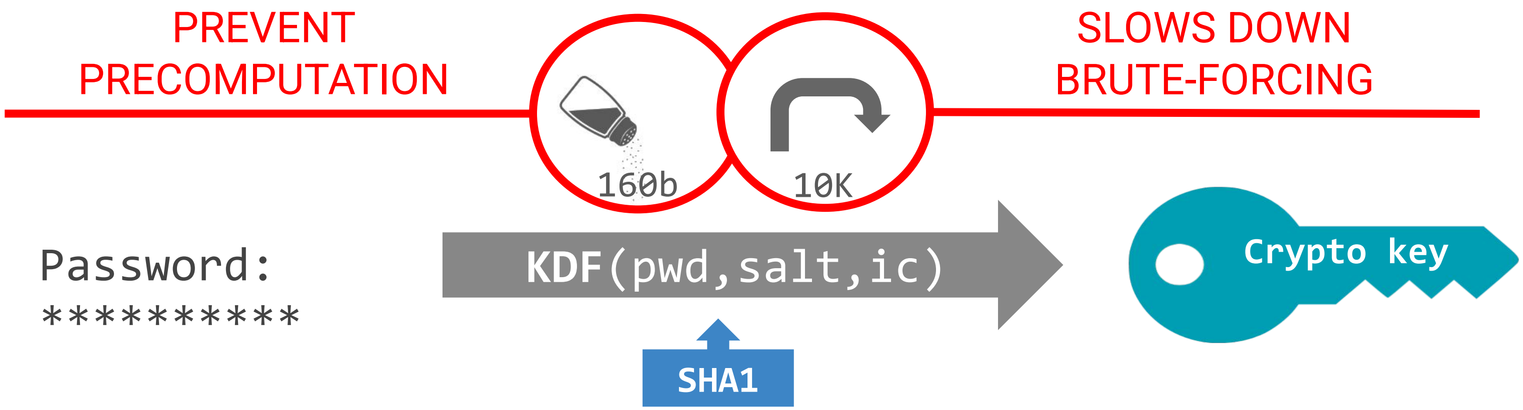
Key Confidentiality ✓
Key Integrity ✓
System Integrity ✓

******

Keystore

- File containing keys and certificates
- <u>Password-protected</u>

Key derivation function (KDF)



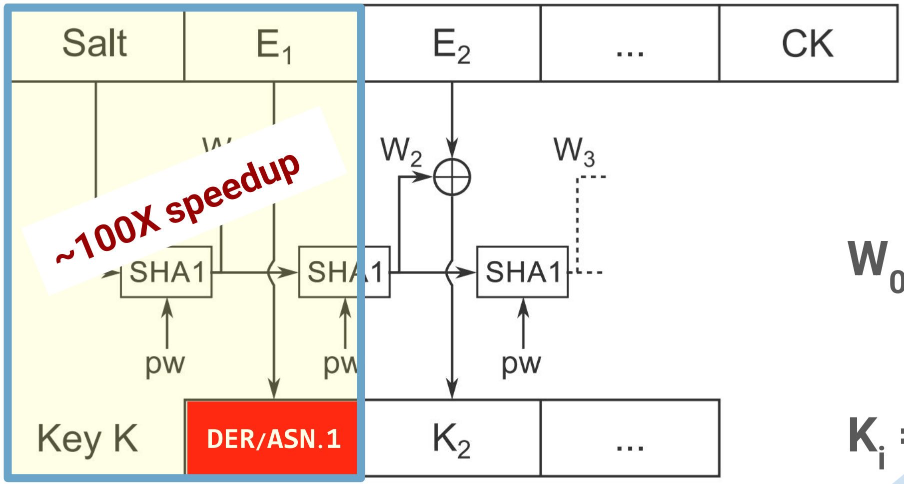
⇒ KDF is similar to password hashing but outputs a crypto key

Oracle JKS Password Cracking

Key Decryption in JKS

8 billions pw/s with one NVIDIAGTX 1080

E = Encrypted Key



W = Keystream

$$W_0 = \text{Salt}$$

$$W_i = \text{SHA1}(pw||W_{i-1})$$

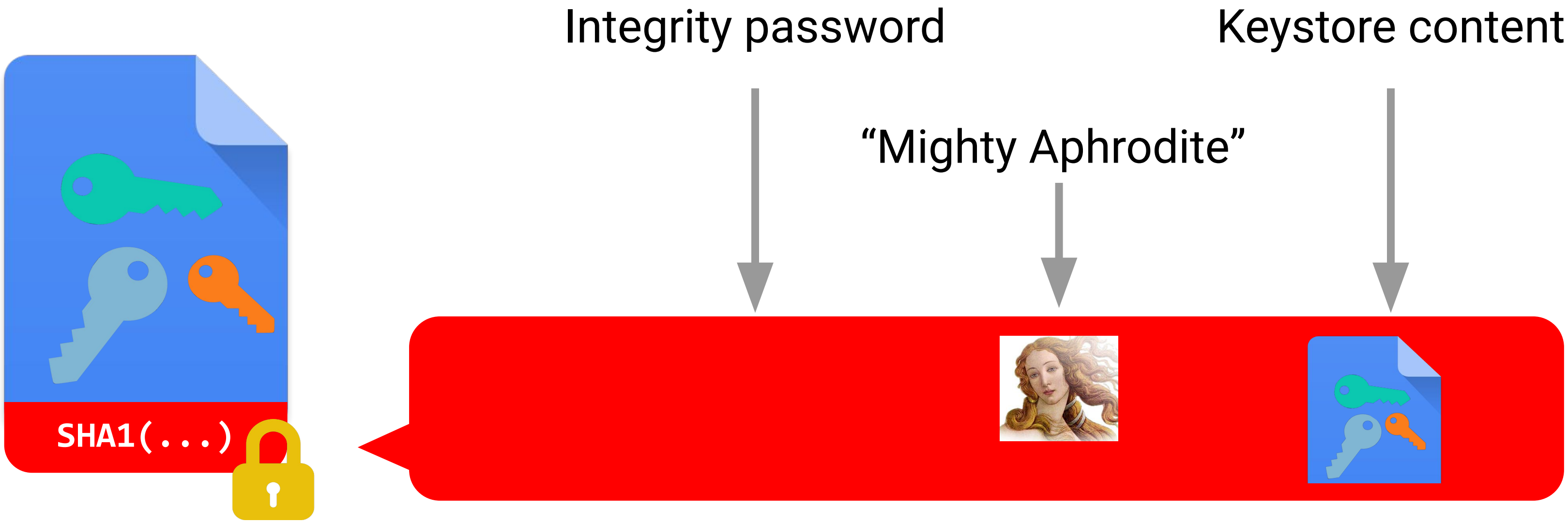
$$K_i = E_i \oplus W_i$$

$$CK = \text{SHA1}(pw||K)$$

K = Decrypted Key

CK = Checksum

JKS/JCEKS Integrity Pwd Cracking



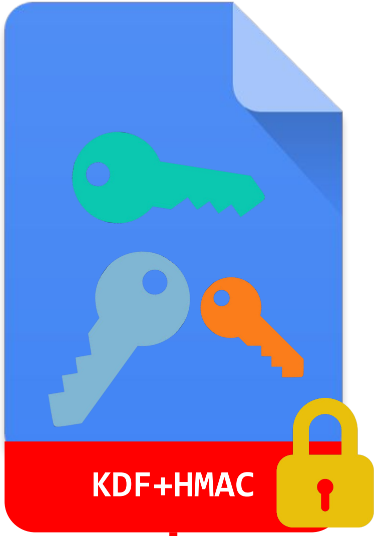
For JKS and JCEKS (Java Cryptography Extension) keystores:

- Efficient **integrity-password bruteforce** (fixed “salt” … precomputation!)
- Watch out when integrity password = confidentiality password!

DoS by Parameters Abuse

Iteration Count = $2^{31}-1$

DoS the application loading the keystore!



Parameters

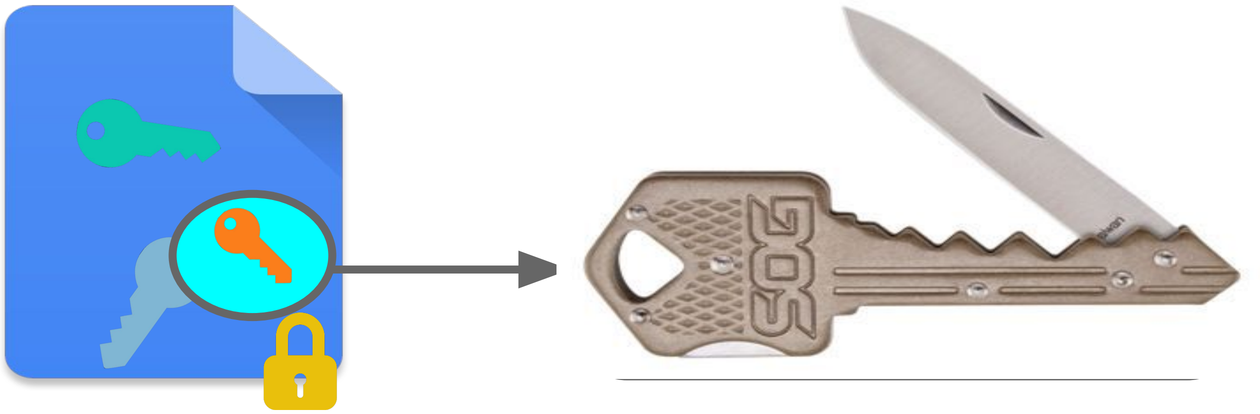
For these keystores:

- Oracle PKCS12
- Bouncy Castle BKS
- Bouncy Castle PKCS12

ASN.1 Str

```
SEQUENCE (3 elem)
   SEQUENCE (2 elem)
     SEQUENCE (2 elem)
       OBJECT IDENTIFIER 1.3.14.3.2.26 sha1 (OIW)
       NULL
     OCTET STRING (20 byte) C9C2AF5A...
   OCTET STRING (20 byte) 7B223BBC...
   INTEGER 1024
```

JCEKS Secret Keys Code Exec

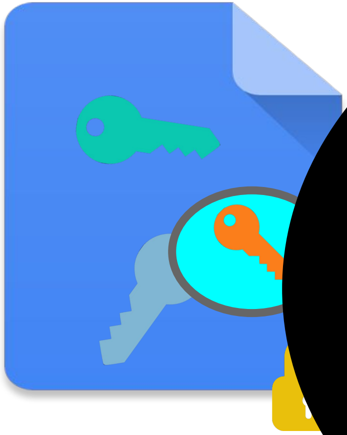


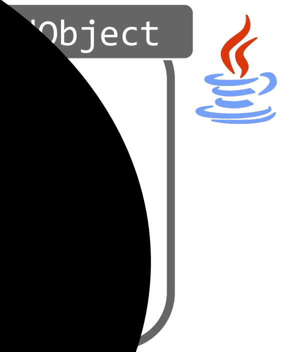
KeyStore Load Mechanism

- **deserialize** each SealedObject
- then perform **Integrity Check**

- **Command execution JDK≤1.7.21 & JDK≤1.8.20**
- **DoS JDK>1.8.20**
- **Fixed Oct 2017 CPU**

JCEKS Code Exec <u>after Decrypt</u>





JCEKS

Rebrand ;)

Java **C**ode **E**xecution **K**ey**S**tore

Deserialize of SecretK

- Extended **classpath**
- Use gadgets from any **3rd-party library**

mmand execution on latest JDK if integrity & key password are known!

Java keystore vulnerabilities (NDSS18)

… 2017
Keystore Analysis

Apr 2017
Discovered attacks, including **code execution**

May 2017
Report to Oracle and BC

Jul 2017
Issues fixed by Oracle

Aug 2017
BC1.58 released fixing some issues

Oct 2017
Oracle CPU
CVE-2017-10345, CVE-2017-10356

Nov 2017
JCEKS code exec, again...

Feb 2018
Full disclosure @NDSS18

Apr 2018
Oracle CPU
CVE-2018-2794

(For more information see the paper and the presentation at NDSS18)

Responses

CVE-2017-10356
CVSS 6.2

- Oracle Keytool, **warning** on JKS/JCEKS
  - `The JCEKS keystore uses a proprietary format. It is recommended to migrate to PKCS12 which is an industry standard format [...]`
- Oracle JCEKS KDF params for PBE
  - from 20 to **200K iterations** (max 5M)
- Oracle PKCS12
  - from 1024 to **50K** iterations for PBE (max 5M)
  - from 1024 to **100K** iterations for HMAC (max 5M)

- Fix(es) to the Oracle JCEKS code execution

- Similar improvements in **Bouncy Castle**

CVE-2017-10345
CVSS 3.1
CVE-2018-2794
CVSS 7.7

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 classes of attacks: *adversarial machine learning*