User Authentication 2

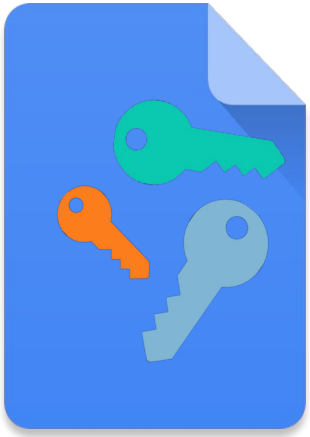
System Security (CM0625, CM0631) 2026-27
Università Ca' Foscari Venezia

Riccardo Focardi
https://www.unive.it/people/focardi
secgroup.dais.unive.it

Università
Ca'Foscari
Venezia

Case study 2: Java keystores

Key Storage



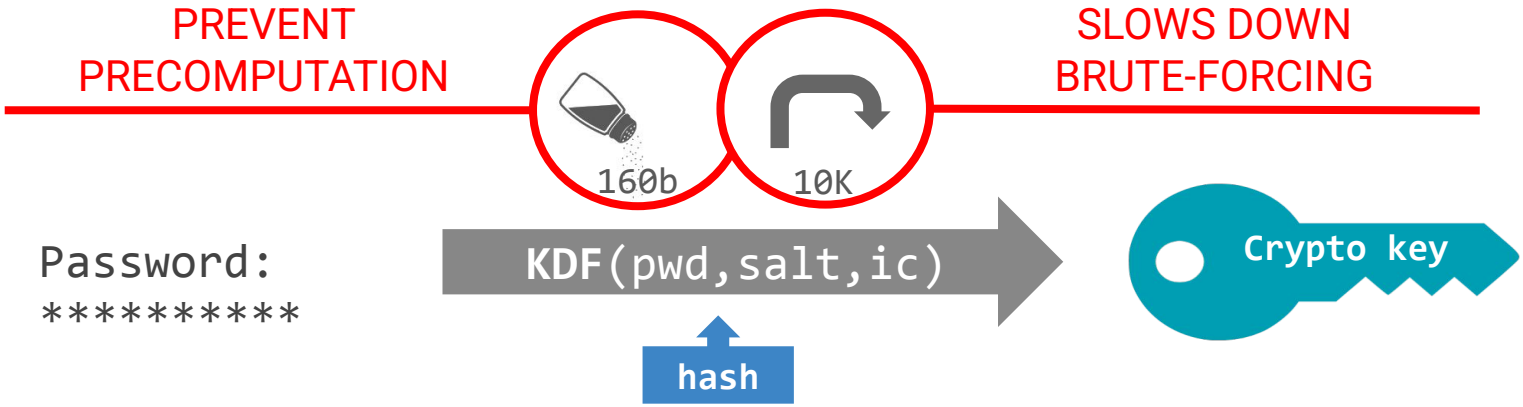
Key Confidentiality ✓
Key Integrity ✓
System Integrity ✓

******

Keystore

- File containing keys and certificates
- Password-protected

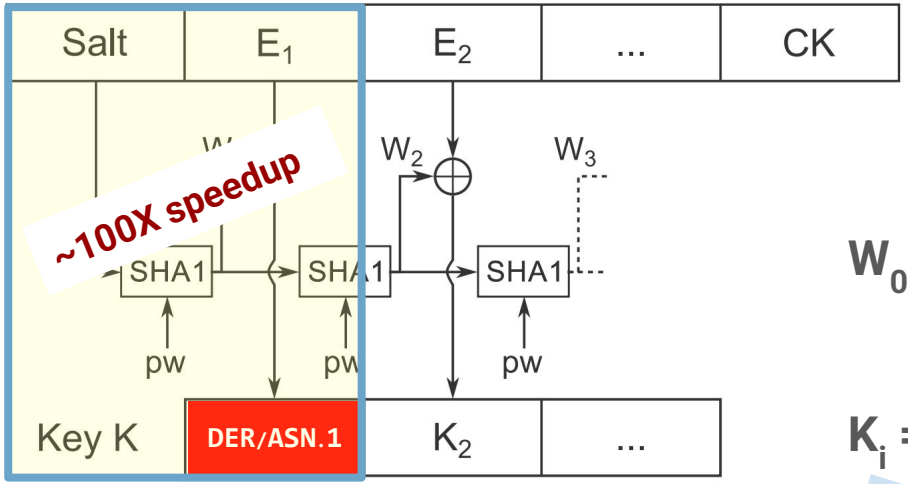
Key derivation function (KDF)



⇒ KDF is similar to password hashing but outputs a crypto key
(has a **salt** and an iteration counter **ic**)

Oracle JKS Password Cracking

Key Decryption in JKS



E = Encrypted Key

W = Keystream

$W_0 = Salt$

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

$K_i = E_i \oplus W_i$

$CK = SHA1(pw||K)$

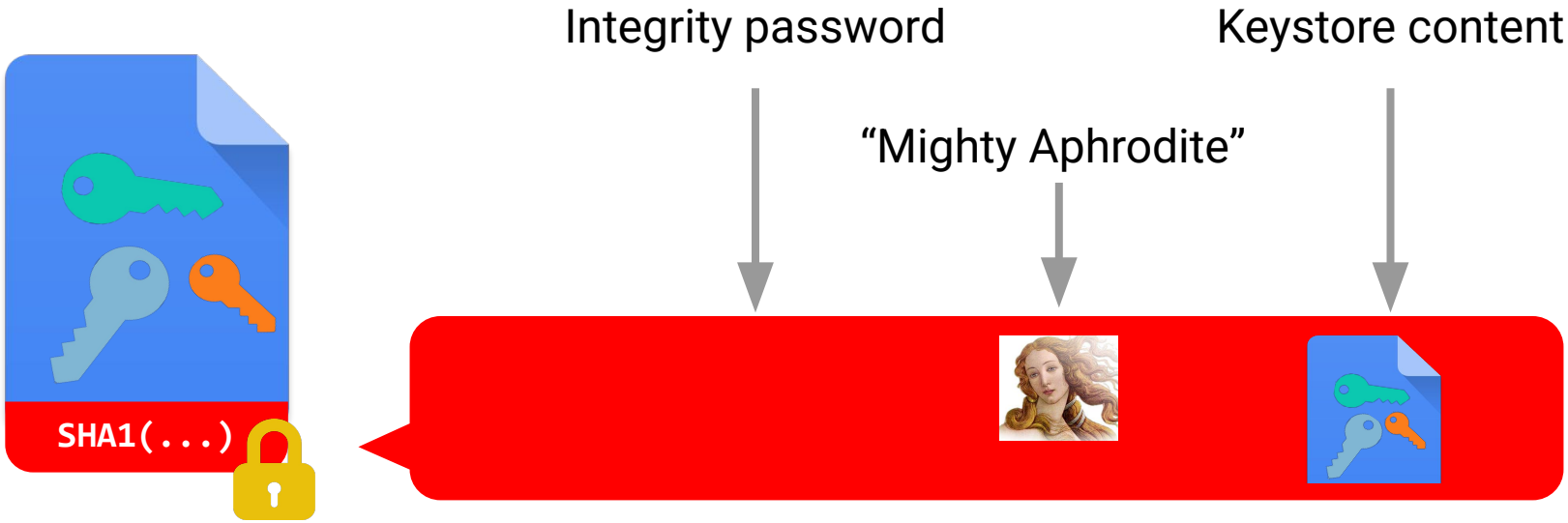
K = Decrypted Key

CK = Checksum

HASHCAT

8 billions pw/s with one NVIDIAGTX 1080

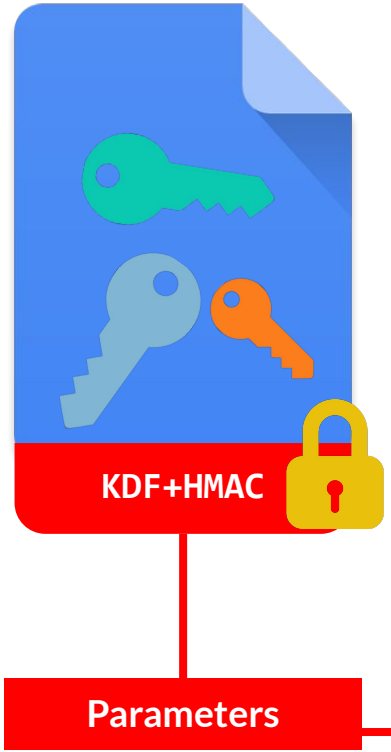
JKS/JCEKS Integrity Pwd Cracking



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



For these keystores:

- Oracle PKCS12
- Bouncy Castle BKS
- Bouncy Castle PKCS12

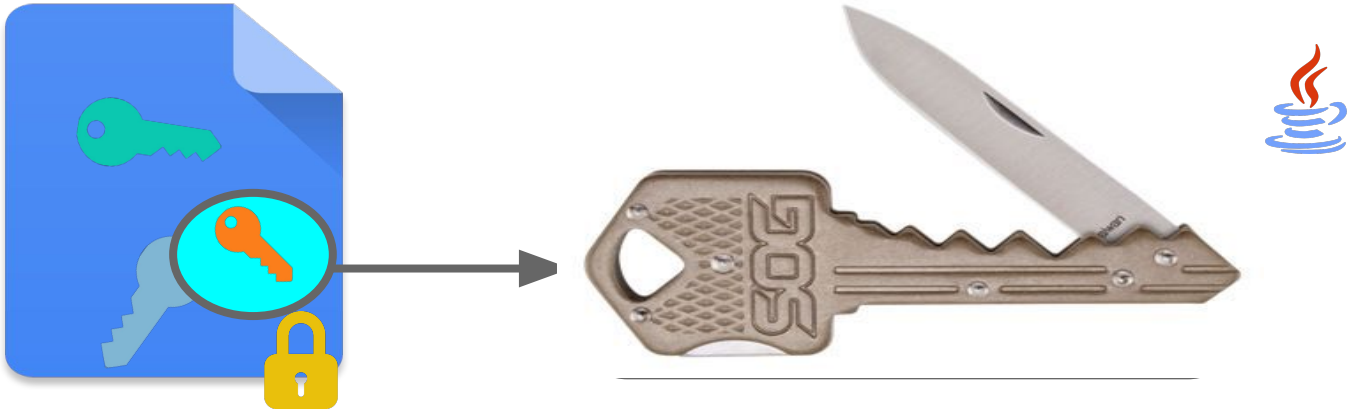
Iteration Count = $2^{31}-1$

DoS the application loading the keystore!

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

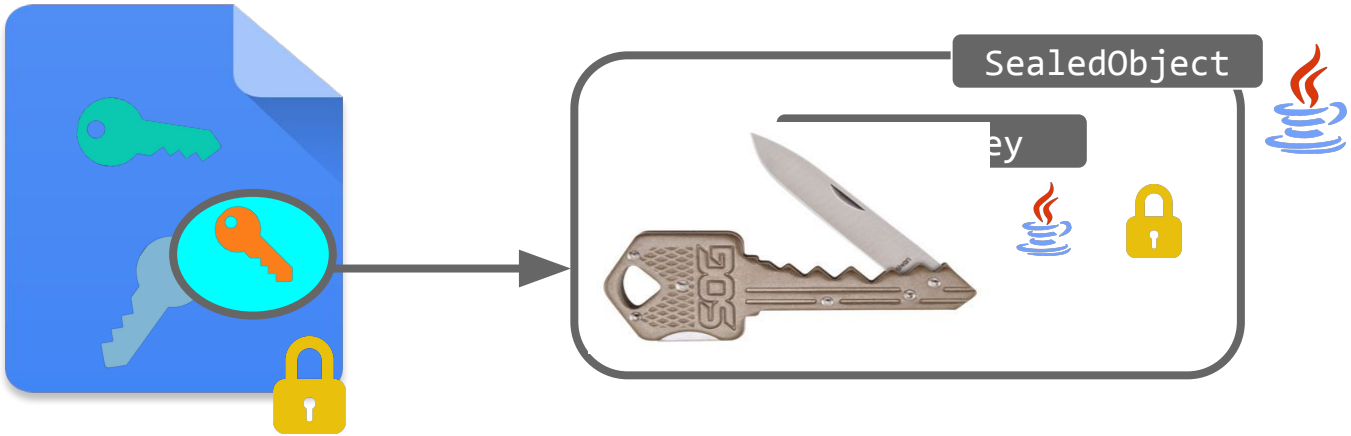


KeyStore Load Mechanism

- **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 after Decrypt

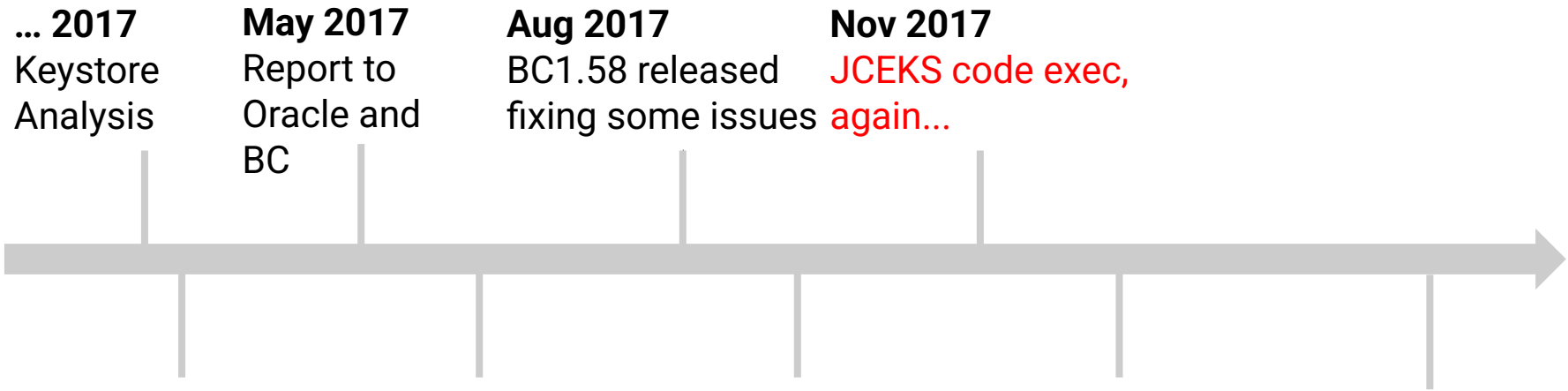


Deserialize of SecretKey

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

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

Java keystore vulnerabilities (NDSS18)



(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)

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

- Fix(es) to the Oracle JCEKS code execution
- Similar improvements in **Bouncy Castle**

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*

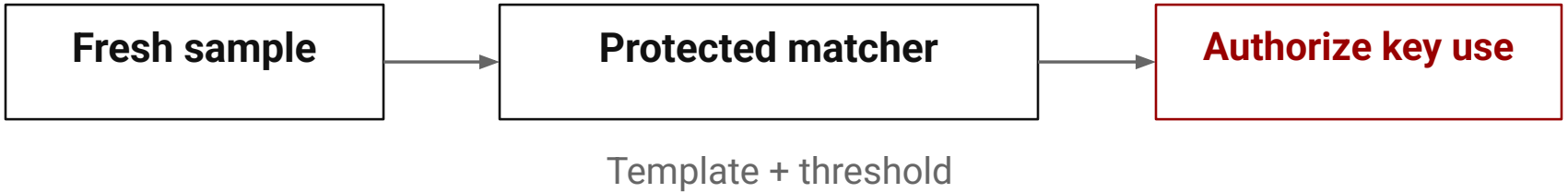
Biometric unlocking

Enrollment

Samples produce a protected feature template T.

Verification

A fresh sample produces features x. Accept when score(x, T) ≥ τ.

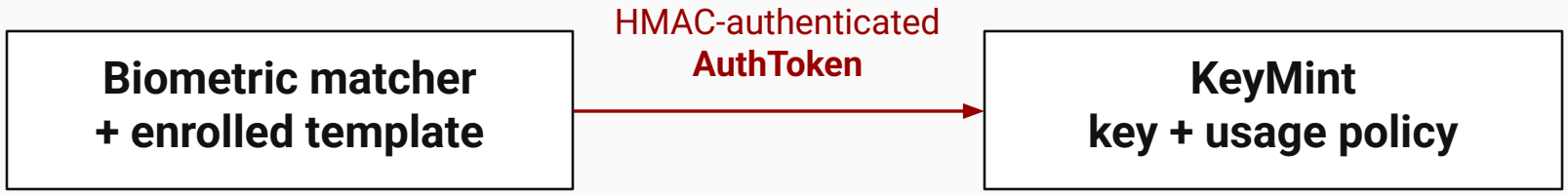


Noisy measurements do not directly reproduce a cryptographic key

Protect the **enrolled template** and **matching logic**, prevent sample **injection** or **replay**, and securely bind successful verification to key use

Biometrics and Android Keystore

Secure environment (Trusted Execution Environment, TEE / secure hardware)



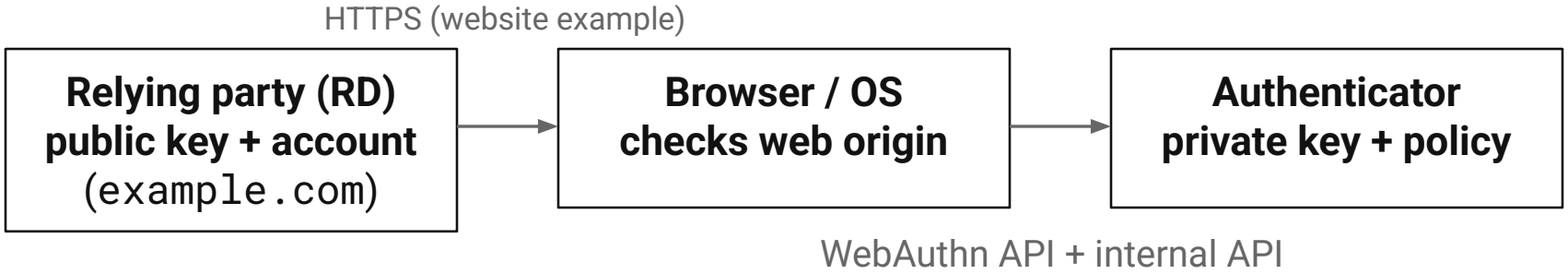
AuthToken: user SID, authenticator type, timestamp, operation challenge, MAC

- Policy can require authentication per operation or within a time window
- The per-boot **HMAC key** stays within secure components, inaccessible to Android services and apps.

- Android forwards the **AuthToken**
- **KeyMint** verifies it and executes the authorized key operation

Passkeys and the keystore model

A **passkey** is a **public–private key** credential used to sign in to a website or app



Local authorization

PIN or biometrics authorizes key use

Remote authentication

A signature proves key possession
The website receives no biometric

Passkey registration

1. Server	Send fresh challenge, RP ID and USER ID for the current account.
2. Browser	Validate RP ID against origin (e.g. `example.com`)
3. Authenticator	Verify user, generate (sk, pk), bind credential to RP + account.
4. Server	Validate registration and store credential ID + pk under the account.

Secure enrollment determines whose key the account will trust

Passkey authentication

The server sends a fresh challenge c, the user authorizes key use

clientDataJSON

type = "webauthn.get"
challenge = base64url(c) (from the server)
origin = "https://login.example.com"

authenticatorData

rpIdHash = SHA-256(RP ID)
flags = …
a counter to prevent replay

σ = Sign_sk(authenticatorData || SHA-256(clientDataJSON)) + USER ID

Server verifies c, origin, RP ID, flags, account mapping and σ

Origin binding and phishing relay

Genuine service

origin: https://login.example.com
RP ID: example.com

Phishing page

origin: https://attacker.example
Relays the genuine challenge c

Request RP ID = example.com

The browser rejects a request from the unrelated phishing origin.

Request RP ID = attacker.example

The genuine passkey is outside scope. The RP also checks signed origin.

Key protection and synchronization

Property	Device-bound	Synced
Availability	Original authenticator	Authorized provider devices
Private-key mobility	No sync / export by design	Encrypted transfer / recovery
Loss of device	Another registered credential or account recovery	Provider recovery / other device
Local key protection	Depends on authenticator	Depends on provider + device

Synced passkeys rely on the **provider's encryption** and secure device enrollment to protect the **private key across devices**

Discussion: what does the server trust?

1. A database leaks public keys and credential IDs

Can an attacker authenticate as Alice? Which secret is missing?

2. A browser session cookie is stolen after a passkey login

Does biometric verification at login prevent session hijacking?

3. The response contains Bob's USER ID and Alice's signature

Which server-side check must reject the response?

References

W3C WebAuthn Level 3
Protocol, signed data, RP ID and origin validation

FIDO Alliance: Passkeys
FIDO credentials, device-bound and synced passkeys

Yubico WebAuthn Developer Guide
Registration, discoverable credentials and user handles

Android: Authentication
Trusted authenticators, AuthTokens and KeyMint

Android Keystore system
Non-exportable keys and user-authentication policies

Android: Biometrics
Biometric strength classes and Keystore integration

Apple: Touch ID security
Local templates and Secure Enclave matching