Security and Cryptography 2021

Mirosław Kutyłowski

X. CLONE DETECTION and AVOIDANCE

problem: a hardware token executing cryptographic protocol can be cloned once the attacker gets access to the internal state of the token with all secrets

Strategies

- no secrets in full control of one party/device (e.g.: distributed generation of keys)
- making clones useless (rapid changes and synchronization)
- immediate detection of active clones

Distributed key generation

- split responsibility for the key quality, at least 2 parties involved
- result:
  i one party learns the key
  ii 2 parties share a key, but nobody has the entire key

Easy case – DL based systems

DH based procedure:

1. device A sends $X_0 = g^{x_0}$ to device B
2. device B sends $X_1 = g^{x_1}$ to device A
3. A responds with x_0 (maybe encrypted with $K = \mathrm{Hash}(X_1^{x_0})$)
4. B computes the public key $K = X_1^{x_0}$ and the private key $x := x_0 \cdot x_1$
5. A can check that the resulting key is K but has no knowledge about x

A version where A and B keep key shares, respectively, x_0 and x_1

Hard case – RSA

necessary to derive 2 prime numbers so that neither A nor B knows any of these primes

trick (from Estonian ID cards)

use 4K-bit numbers that have 4 prime factors instead of 2

backwards comp.

observation: the same algebra as for the original RSA show that if

$$e \cdot d = 1 \bmod \ldots.$$

then

$$(m^d)^e = m \mod n$$

Smart ID key generation

1. App generates a 2048-bit RSA key pair with the private key (n_1, d_1) and public key (n_1, e)
2. App chooses d_1' at random

   $e \cdot d_1 = 1 \mod \text{order } \mathbb{Z}_n^*$

3. App computes $d_1'' = d_1 - d_1'$
4. App encrypts d_1' with its PIN, stores the ciphertext and deletes its plaintext
5. App deletes plaintext of d_1 (and information leading to factors of n_1)
6. App sends n_1, e, d_1'' to SecureZone
7. SecureZone generates the 2048-bit RSA key pair with private key (n_2, d_2) for public key (n_2, e)
8. SecureZone computes α, β so that

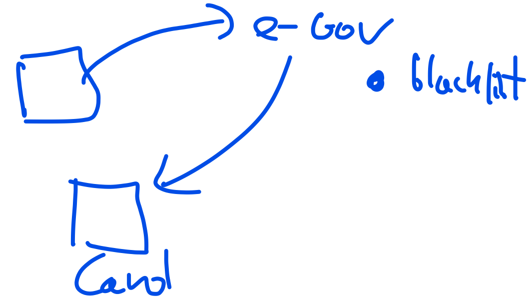
   $\beta = \frac{1}{n_2} \mod n_1$

   $$\alpha \cdot n_1 + \beta \cdot n_2 = 1 \mod n_1$$

   (Euclidean algorithm for integers, it works as n_1 and n_2 are coprime whp).
9. SecureZone computes the user's public modulus $n = n_1 \cdot n_2$

public key of a user is (n, e)

distributed "RSA" signature generation for M



1. App asks for the PIN and decrypts the ciphertext of d_1'
2. App computes m - encoding of M
3. App computes $s_1' := m^{d_1'} \bmod n$ and sends it to Smart-ID Server
4. Smart-ID Server computes m - encoding of M
5. Smart-ID Server computes $s_1'' = m^{d_1''} \bmod n_1$
6. Smart-ID Server computes $s_1 = s_1' \cdot s_1'' \bmod n$

   (so $s_1 = m^{d_1} \bmod n_1$) $s_1' \cdot s_1'' = m^{d_1'} \cdot m^{d_1''} = m^{d_1'+d_1''} = m^{d_1} \ldots$
7. Smart-ID Server computes $s_2 = m^{d_2} \bmod n_2$
8. Smart-ID Server computes

$$S := \beta \cdot n_2 \cdot s_1 + \alpha \cdot n_1 \cdot s_2 \bmod n$$

   (by ChRT to get S such that $S = s_1 \bmod n_1$ and $S = s_2 \bmod n_2$

output: signature S

Verification

as for RSA: checking that $S^e = m \bmod n$

$$S^e = m \bmod n \quad \text{iff} \quad S^e = m \bmod n_1 \;\text{ and }\; S^e = m \bmod n_2$$

$$s_1^e = m \bmod n_1 \;\text{ and }\; s_2^e = m \bmod n_2$$

$$(m^{d_1})^e = m \bmod n_1 \;\text{ and }\; (m^{d_2})^e = m \bmod n_2$$

$$S^e = m \bmod n_1 \qquad S^e = m \bmod n_2$$

Security concept

in order to create a signature alone:

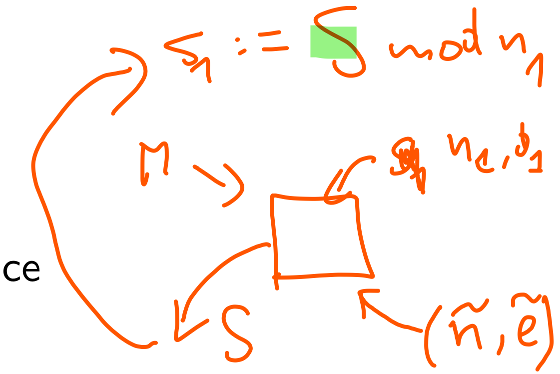
- App would need to create $m^{d_2} \bmod n_2$ – impossible if the original RSA signature is unforgeable
- Smart-ID server would need to create $m^{d_1} \bmod n_1$. It knows n_1 but the exponent d_1'' is random, so cannot help to forge an RSA signature for modulus e

---

Conclusion

distributing private key can work

whereas an adversary can typically clone at most one device



Clone detection concepts

1. hide invissible characteristics in the device that may be used to fish out clone's signatures post factum

2. discourage to use clones: key compromise in case of clone usage

3. fluctuation of distributed key

Key fluctuation

works for RSA, EdDSA, Schnorr, ...

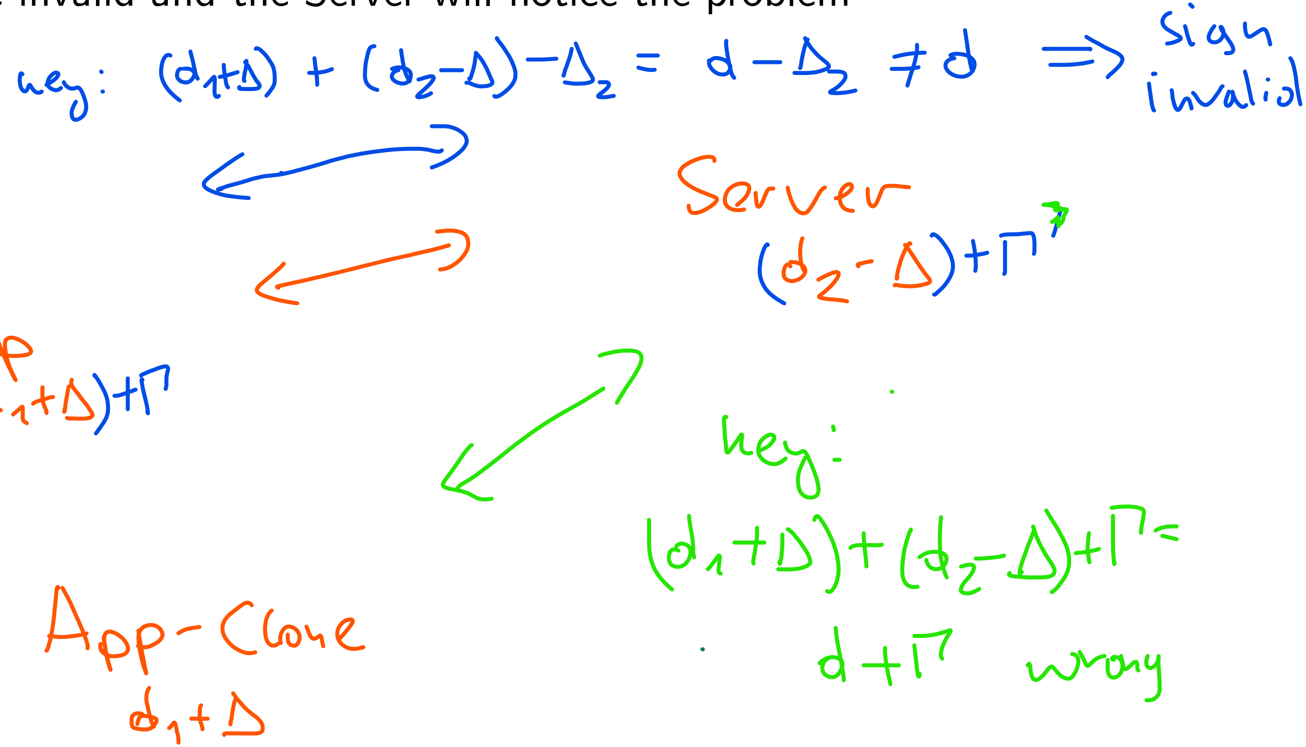
fluctuation (example for plain RSA)

- App holds d_1, Server holds d_2
- signature creation:
  - i an integer Δ is negotiated
  - ii App updates: $d_1 := d_1 - \Delta$
  - iii Server updates $d_2 := d_2 + \Delta$

(computations over integers, as the group order is unknown)

Security concept of key fluctuation

- App and Server must be synchronized
- If App_1 and App_2 are clones, then App_1 de-synchronizes App_2: if it attempts to sign, then the signature will be invalid and the Server will notice the problem



Tokens - example Smart-ID

Clone detection works thatnks to the following nonce (original Estonian description):

one-time password *– created by Smart-ID Core in the end of each operation (incl. initialization) and valid until the completion of next.*

retransmit nonce *– created in the beginning of each operation by Smart-ID App, the same value must be used when Smart-ID App retries messages to Smart-ID Core, related to the same operation.*

freshness token – created by Smart-ID Core before each submission operation from Smart-ID App to Smart-ID Core. Ensures that state-changing operations get executed in the order client issued them (although some may be missing from between).

sign(M || R')

Linking – microTESLA ...

at session k:

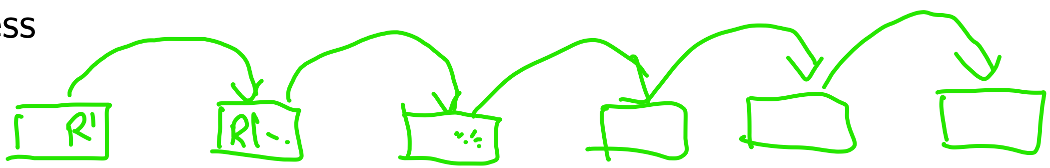
i. A chooses R at random, $R' := \mathrm{Hash}(R)$ (or an HMAC of R is MAC key shared)

ii. A attaches R' to the current transmission

at session $k+1$:

i. A authenticates himself with R

$\Rightarrow$ if at some moment a clone is created and does not hijack synchronization with the server, then it is useless



Detection of active clones

idea: clone may emerge, but their holder will never use them without revelaling compromise

two examples:

1. failstop signatures
2. commitments

Failstop signatures

Domain Parameters and Keys:

- G_q – a group of a prime order q such that DLP is hard in G_q
- $g, h \in G_q$ be such that nobody should know $\log_g h$
- one-time secret $\mathrm{SK} = (x_1, x_2, y_1, y_2)$
- one-time public key $\mathrm{PK} = (g^{x_1} h^{x_2}, g^{y_1} h^{y_2})$

h = Hash(stockex ... 1.1.2022)

Failstop one-time signature

- $\mathrm{Sign}(\mathrm{SK}, m) = (\sigma_1(\mathrm{SK}, m), \sigma_2(\mathrm{SK}, m))$ where
- $\sigma_1(\mathrm{SK}, m) = x_1 + m \cdot y_1 \bmod q$

  $\sigma_2(\mathrm{SK}, m) = x_2 + m \cdot y_2 \bmod q$

Failstop signature verification

if $\mathrm{PK} = (p_1, p_2)$ then the signature is valid iff

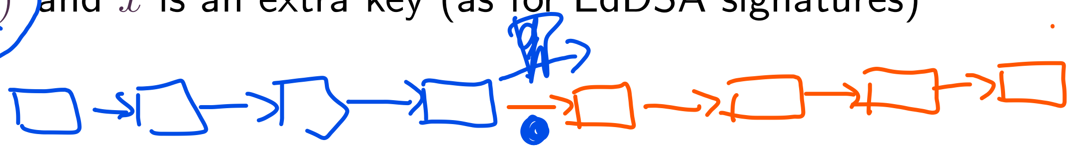
$$p_1 \cdot p_2^m = g^{\sigma_1} \cdot h^{\sigma_2}$$

Security concept

- there are q solutions for σ_1, σ_2
- an adversary breaking p_1, p_2 may have valid keys, can use them, but then the legitimate user can derive $\log_g h$

Commitment to ephemeral values

- signature i contains a commitment to $r_{\text{next}} = g^{k_{\text{next}}}$ used in the next signature. E.g., the signature is over $M \| \text{Hash}(r_{\text{next}})$ instead of M
- the next signature uses $r = r_{\text{next}}$
- in order to remember r_{next} one can design a scheme where $r_i = g^{k_i}$ where $k_i := \text{Hash}(x, i)$ and x is an extra key (as for EdDSA signatures)



Situation:

- the ith signature created by a clone and the ith signature created by the original device - use the same k_i
- the same k_i for different messages $\Rightarrow$ secret key gets exposed
- so: using a clone reveals the fact that the key is compromised