Introduction to Cryptography

F. Koeune – O. Pereira

MAT2450 – Lecture 9

Part I

RSA signature

The Group $\mathbb{Z}_N^*$

Define $\mathbb{Z}_N^* := \{a \in \{1, \ldots, N-1\} \,|\, \gcd(a, N) = 1\}$

$\mathbb{Z}_N^*$ with multiplication mod N forms a (commutative) group

- $\exists 1_{\mathbb{G}}$ s.t.: $\forall g \in \mathbb{Z}_N^* : 1_{\mathbb{G}} \cdot g = g \cdot 1_{\mathbb{G}} = g$ (1 is fine)
- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- $a \cdot b = b \cdot a$
- $\forall g \in \mathbb{Z}_N^*, \exists g^{-1} \in \mathbb{Z}_N^* : g \cdot g^{-1} = 1$ (as $\gcd(g, N) = 1$)

Example: $\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$

- $3 \cdot 9 = 7$ $\quad 9 \cdot 9 = 1$ $\quad 3 \cdot 7 = 1$

Euler ϕ function

Define $\phi(N) := |\mathbb{Z}_N^*|$

- if N is prime, then $\phi(N) = N - 1$
- if $N = pq$ (where p, q are prime), then
  - $p - 1$ elements of $[1, N-1]$ are divisible by q
  - $q - 1$ elements of $[1, N-1]$ are divisible by p
  - So, $\phi(N) = (N-1) - (p-1) - (q-1) = (p-1)(q-1)$
- If $N = \prod_i p_i^{e_i}$ with distinct prime p_i, then $\phi(N) = \prod_i p_i^{e_i - 1}(p_i - 1)$

$\phi(N)$ is easy to compute if prime factors of N are known

Euler ϕ function

We have

- $\forall g \in \mathbb{Z}_N^*, g^{\phi(N)} = 1$ (Fermat-Euler)

Corollary:

- If $ed = 1 \bmod \phi(N)$, then $g^{ed} = g^{1+k\phi(N)} = g$

$\Rightarrow$ Basis for the *RSA Permutation*

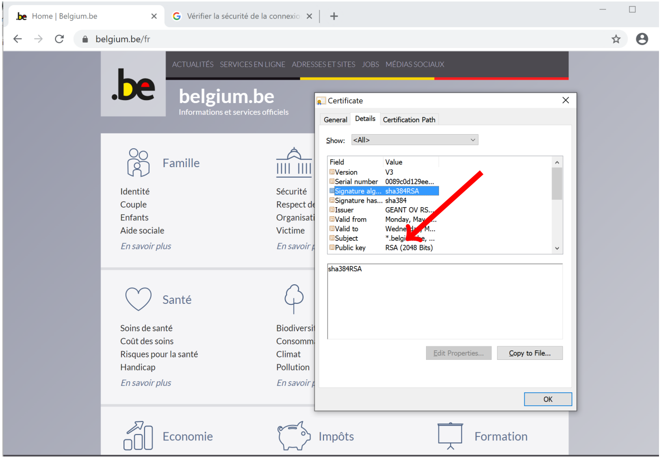
The RSA Permutation

Shamir - Rivest - Adleman

The RSA Permutation

Clifford Cocks

The RSA Permutation



The RSA permutation

Define $f_e(x) := x^e$.

Remember that:

$$\forall g \in \mathbb{G} \text{ with } m := |\mathbb{G}|, \text{ if } \operatorname{ord}(g) = i, \text{ then } i|m$$

So, if $\gcd(e, \phi(N)) = 1$ then f_e is a permutation.

$$(\text{indeed, } a^e = b^e \Rightarrow (ab^{-1})^e = 1 \Rightarrow a = b)$$

And, if $ed = 1 \bmod \phi(N)$, then f_d is its inverse.

$\Rightarrow$ Could be used as a *trapdoor permutation*.

Trapdoor permutation

Idea: a permutation f_e so that:

- Computing $f_e(x)$ is easy for everyone knowing e
  ("easy" = can be done in polynomial time)
- Computing $f_e^{-1}(x)$ is difficult (even knowing e)
- But there exists a trapdoor d allowing computing $f_e^{-1}(x) = f_d(x)$ easily

Could be used to build

- Public key encryption schemes
- Signature schemes

Is RSA a trapdoor permutation?

We have

- Computing g^e is easy (square & mult.)
- If d is known, computing g^d is easy
- If d is unknown...

Computing d given e

- If $\phi(N)$ is known,
  - then finding d s.t. $e.d = 1 \bmod \phi(N)$ is easy (Euclid)
- If $N = p \cdot q$ then:

$$\text{Computing } \phi(N) \Leftrightarrow \text{factoring } N$$

  - Suppose $\langle N, \phi(N) \rangle$ is given
  - $\phi(N) = (p-1)(q-1) = N - (p+q) + 1$
    $\Rightarrow p + q = N - \phi(N) + 1$
- *Fact:* computing d from (N, e) is as hard as factoring
- But we need something stronger:
  Given $[m^e \bmod N]$, m should be hard to compute!

The RSA problem

RSA-inv$_{\mathcal{A}}$ experiment

1. $\langle (N, e), (N, d) \rangle \leftarrow \mathrm{Gen}(1^n)$
2. $y \leftarrow \mathbb{Z}_N^*$
3. $x \leftarrow \mathcal{A}(N, e, y)$
4. Define RSA-inv$_{\mathcal{A}}(n) := 1$ iff $x^e = y \mod N$

Assumption:

- For every PPT $\mathcal{A}$, there is a negligible ϵ s.t.:

$$\Pr[\text{RSA-inv}_{\mathcal{A}}(n) = 1] \leq \epsilon(n)$$

The RSA problem is

- believed to be hard
- not known to be equivalent to factoring

Selecting RSA modulus N

We need N to be hard to factor

- else $\phi(N)$ is easy to compute from N

How to select N?

- Small factors make life easier
  Factors of 10000000000 or of 256000 are easy to find...
  But can you factor 91? 221? 9701?
- Best choice seems to choose N as the product of two large factors

Selecting RSA modulus N

How to select N as $p \cdot q$?

- We need large primes...

How large?
Factoring records include primes such as[1]

$$
\begin{gathered}
64135289477071580278790190170577389084825014\backslash \\
42943447208116859632024532344630238623598752\backslash \\
683477087376619255856946397988533 67
\end{gathered}
$$

[1] See, e.g., https://en.wikipedia.org/wiki/Integer_factorization_records#Numbers_of_a_general_form

Building a signature scheme with RSA

Simple signature scheme proposal ("textbook RSA")

- $\mathrm{Gen}(1^n) := \langle (N, e), (N, d) \rangle$ with $|p| = |q| = n$
- $\mathrm{Sign}_{(N,d)}(m) := [m^d \bmod N]$
- $\mathrm{Vrfy}_{(N,e)}(m, \sigma) := [\sigma^e \stackrel{?}{=} m \bmod N]$

Does that "work"?

- Yes: $[(m^d)^e = m \bmod N]$

Is this secure?

Textbook RSA

Is this secure?

Certainly not!

- No-message attack
  - Take σ at random
  - Compute $m := [\sigma^e \bmod N]$
  - (m, σ) is a forgery

(Of course, over an "uncontrolled" message, but we already discussed that question)

Textbook RSA

Is this secure?

Certainly not!

- Signature combination
- Suppose $\mathcal{A}$ wants to forge a signature on m
  - $\mathcal{A}$ chooses m_1 at random and obtains signature σ_1
  - $\mathcal{A}$ computes $m_2 := [m/m_1 \bmod N]$ and obtains signature σ_2
  - Now, $\sigma := [\sigma_1\sigma_2 \bmod N]$ is a valid signature on m

$$\sigma^e = (\sigma_1.\sigma_2)^e = (m_1^d.m_2^d)^e = m_1^{ed}.m_2^{ed} = m_1.m_2 = m \pmod{N}$$

Hashed RSA

Basic idea: hash the message before applying RSA, i.e.

$$\sigma(m) := [H(m)^d \bmod N]$$

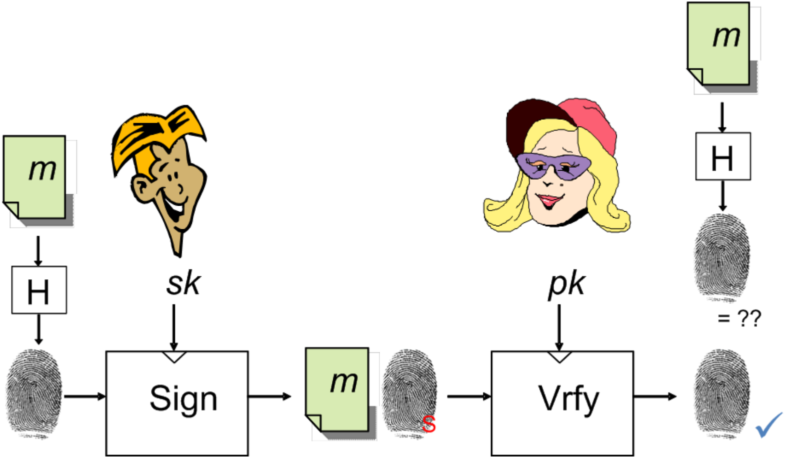
Verification is simple:

$$[\sigma^e \stackrel{?}{=} H(m) \bmod N]$$

Of course, H must be collision resistant

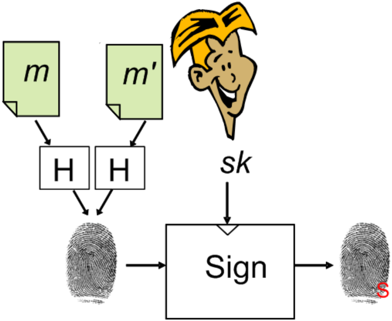
- Otherwise, one could forge a signature for m from the signature of m' s.t. $H(m) = H(m')$

Hash-then-sign



Collision on hash function

If H were not collision resistant. . .



Does this seem to work?

At least, it thwarts the attacks we had found so far:

- No-message attack:
  - $\mathcal{A}$ would need to find m s.t. $H(m) = [\sigma^e \bmod N]$
  - Difficult if H is pre-image resistant
- Signature combination
  - $\mathcal{A}$ would need to find m, m_1, m_2 s.t. $H(m) = [H(m_1).H(m_2) \bmod N]$
  - *Seems* difficult for traditional hash functions (e.g., SHA-2, SHA-3)

Can we prove that this works?

No

There is no expected property of H for which hashed RSA signatures can be proven secure in the sense of our definition

We can prove the security of (a small variant of) this kind of construction, but only in the random oracle model

RSA-FDH (full-domain hash)

Consider textbook RSA as defined previously, and let H be a hash function whose range can be set to $\mathbb{Z}_N^*$ $^{(*)}$

Define the **RSA-FDH** scheme Π as follows:

- $\mathrm{Gen}: (N, e, d) \leftarrow \mathsf{RSA}(1^n)$
- Sign: on input (N, d) and $m \in \{0,1\}^*$, output $\sigma := H(m)^d \bmod N$
- Vrfy: on input (N, e), $m \in \{0,1\}^*$ and σ, output 1 iff $\sigma^e \overset{?}{\equiv} H(m) \bmod N$

Theorem: If the RSA problem is hard, then Π is EUF-CMA in the ROM

$^{(*)}$ For simplicity, we will ignore this issue

Idea (1)

We will consider a $\mathcal{A}$ that can break the scheme and turn it into a $\mathcal{A}'$ that can solve the RSA problem

Behaviour of $\mathcal{A}$

- Receives (N, e) and must find (m, σ) s.t. $\sigma^e \equiv H(m) \bmod N$
- $\mathcal{A}$ has access to
  - A random oracle H providing $H(m_i)$ for m_i chosen by $\mathcal{A}$
  - A signature oracle $\mathrm{Sign}_{(N,d)}$ providing $\sigma_i := H(m_i)^d \bmod N$ for m_i chosen by $\mathcal{A}$

Behaviour of $\mathcal{A}'$

- Receives (N, e, y^*) and must find x s.t. $x^e \equiv y^* \bmod N$

Idea (2)

As usual, $\mathcal{A}'$ will run $\mathcal{A}$ as a subroutine

- $\mathcal{A}'$ receives (N, e, y^*) and transmits (N, e) to $\mathcal{A}$
- $\mathcal{A}'$ answers queries of $\mathcal{A}$, acting both as H and $\mathrm{Sign}_{(N,d)}$
- $\mathcal{A}'$ receives the forgery (m, σ) from $\mathcal{A}$ and must turn it into an e-th root x s.t. $x^e \equiv y^* \bmod N$

Idea (3)

Problem: $\mathcal{A}'$ must answer the signature requests of $\mathcal{A}$

- $\mathcal{A}$ outputs m_i and expects in return $\sigma_i = H(m_i)^d$
- But $\mathcal{A}'$ does not know the private key d !

Solution: use the random oracle's programmability to "cheat"

- When $\mathcal{A}'$ receives a request for m_i (either hash or sign)
  - Choose a random value σ_i
  - Compute $y_i := \sigma_i^e \bmod N$
  - Decide that $H(m_i) := y_i$
- But that works only *provided $\mathcal{A}$ does not see any difference with regular random oracle output*
- As σ_i is uniformly distributed and RSA is a permutation, y_i will be uniformly distributed $\Rightarrow$ OK

Idea (4)

Problem: How can $\mathcal{A}'$ use $\mathcal{A}$ to compute the e^{th}-root of the specific value y^* ?

Solution: again, "cheat"

- As we are in the random oracle model, the only way for $\mathcal{A}$ to learn the value of $H(m_i)$ for any m_i is to ask it to H
- wlog, we can thus assume that all messages processed by $\mathcal{A}$ (requests and final forgery) have been submitted to H
- Let us make sure that one of these requests has y^* for answer
- i.e. for one (randomly-chosen) hash request m, $\mathcal{A}'$ will answer $H(m) := y^*$

Idea (5)

If we "get lucky", the message m involved in the forgery (m, σ) output by $\mathcal{A}$ will be the one we choose (and then we win, as σ is a e^{th}-root of y^*)

- By assumption, there were only a polynomial number $q(n)$ of queries to the random oracle
- We have thus a $1/q(n)$ chance to "be lucky", which is non-negligible

Idea (6)

Of course, $\mathcal{A}'$ must act in a coherent way

- If the hash of a message is requested several times (either implicitly or explicitly), $\mathcal{A}'$ must provide coherent answers

  Explicitly direct request to random oracle

  Implicitly signature request

- $\mathcal{A}'$ will do so by maintaining a table of requests-answers

Formal proof (1)

For simplicity, let's assume that:

- $\mathcal{A}$ never makes the same random oracle request twice
- If $\mathcal{A}$ requests the signature of a message m_i, then it has previously queried $H(m_i)$
- If $\mathcal{A}$ outputs (m, σ), then it has previously requested $H(m)$

(No loss of generality: this will just make the proof easier to read)

Formal proof (2)

Let $\mathcal{A}$ be a PPT adversary breaking Π with probability ϵ, and let q be the (polynomial) number of queries made by $\mathcal{A}$.

We define $\mathcal{A}'$ as follows:

1. $\mathcal{A}'$ is given (N, e, y^*) as input
2. Choose $j^* \leftarrow \{1, \ldots, q\}$
3. Transmit (N, e) to $\mathcal{A}$
4. Store a (initially empty) table of triples $(\cdot, \cdot, \cdot)$
   - Entry (m_i, σ_i, y_i) means that $\mathcal{A}'$ has set $H(m_i) = y_i$ and $\sigma_i^e \equiv y_i \bmod N$
5. When $\mathcal{A}$ makes its ith query $H(m_i)$, answer as follows
   - If $i = j^*$, return y^*
   - Else, choose $\sigma_i \leftarrow \mathbb{Z}_N^*$, compute $y_i := \sigma_i^e \bmod N$, return y_i and store (m_i, σ_i, y_i) in the table

Formal proof (3)

6. When $\mathcal{A}$ requests a signature on m',
   - Let i be the index s.t. $m' = m_i$ in the table
   - If $i = j^*$ return *failure*
   - Else return σ_i as stored in the table
7. When $\mathcal{A}$ returns forgery (m, σ), check whether $m = m_{j^*}$ and $\sigma^e = y^*$
8. If yes, output σ, otherwise, output *failure*

Observations (1)

- $\mathcal{A}'$ runs in polynomial time
- When the guess j^* is correct, the view of $\mathcal{A}$ is distributed identically to the view of $\mathcal{A}$ in experiment $\text{Sig-forge}_{\mathcal{A},\Pi}(n)$:
  - The answer to query $H(m_{j^*})$ is answered with the value y^*, chosen uniformly at random
  - Each answer to a query with $i \neq j^*$ is generated by choosing σ_i uniformly at random and computing $y_i = \sigma_i^e \bmod N$
    - Since RSA is a permutation, y_i is distributed uniformly at random
  - j^* is independent on the view of $\mathcal{A}$
    - Unless $\mathcal{A}$ requests for the signature of m_{j^*}
    - But this cannot happen when the guess is correct, otherwise $\mathcal{A}$ wouldn't output a valid forgery

Observations (2)

$\mathcal{A}'$ wins each time the guess j^* is correct and $\mathcal{A}$ wins

- j^* was chosen at random (and independently of $\mathcal{A}$) among q possible values
- Thus, $\mathcal{A}'$ wins with probability ϵ/q
- If the RSA problem is hard, ϵ/q must be negligible
- Since q is polynomial, ϵ must be negligible as well

Encrypting with RSA

RSA can also be used as a public key encryption scheme.

But the naive option $\mathrm{Enc}_{(N,e)}(m) := m^e \bmod N$ is bad (why?)

Padded RSA:

- Suppose $|m| \in \{0,1\}^{\frac{|N|}{2}-2}$ and $r \leftarrow \{0,1\}^{|N|-|m|-1}$.
  $\mathrm{Enc}_{(N,e)}(m) := [(r||m)^e \bmod N]$
- $\mathrm{Dec}_{(N,d)}(c) := [c^d \bmod N]$ with random padding removed

This is believed to be CPA-secure if the RSA problem is hard

- This was/is a standard way of using RSA (PKCS #1 v.1.5 `http://www.rsa.com/rsalabs`)
- A more sophisticated padding (OAEP) provides CCA-security (PKCS #1 v.$\geq$2.0)

Part II

Certificates and PKI

Certificates and PKI

Asymmetric encryption allows sending a secret msg to Bob...
Digital signature allows verifying that Bob wrote a message...

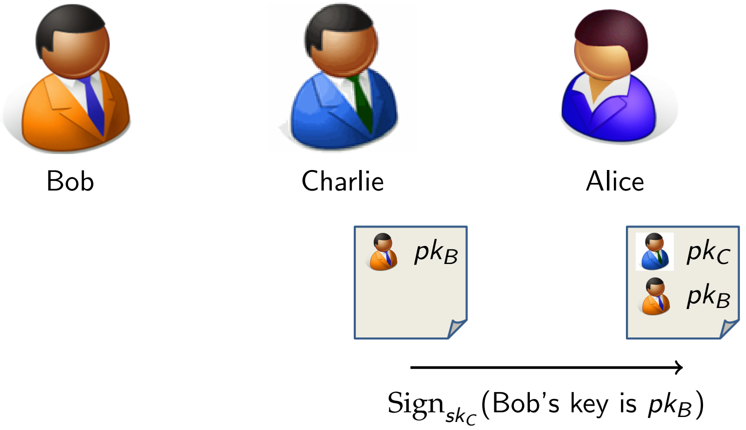
...provided I know Bob's public key

But how can I be sure it is Bob's key?

This is the goal of a *Public Key Infrastructure (PKI)*

- Transmit keys in a trustworthy manner

Transmitting trust in keys



Certificates and PKI

Sign_{sk_C}("Bob's key is pk_B") is called a *certificate for Bob's key issued by Charlie* and denoted $\mathrm{cert}_{C \to B}$

Principle

- Bob generates a key pair pk, sk
- Bob meets Charlie and convinces him that he is Bob and that pk is his public key
- Charlie gives $\mathrm{cert}_{C \to B}$ to Bob
- Bob can now use $\mathrm{cert}_{C \to B}$ to introduce himself to anyone who knows Charlie

Remark: no need for Charlie to know Bob's *secret* key

Certificates and PKI

Alice must be convinced

- That Charlie's key is pk_C
- That Charlie is honest
- That Charlie *does* check Bob's identity before issuing $\mathrm{cert}_{C \to B}$

$\Rightarrow$ Alice considers Charlie as a *certification authority (CA)*

Charlie

- Asserts that Bob's public key is pk_B
- But not that Bob is trustworthy in any way

Certificates and PKI

The certification relationship can be chained

- Bob provides Alice with $pk_B, \mathrm{cert}_{C \to B}, \mathrm{cert}_{D \to C}, \mathrm{cert}_{E \to D}$
- If Alice
  - Has a copy of pk_E, and knows it is authentic
  - And knows that C, D, E are good CAs

  Then she can conclude that Bob's public key is pk_B

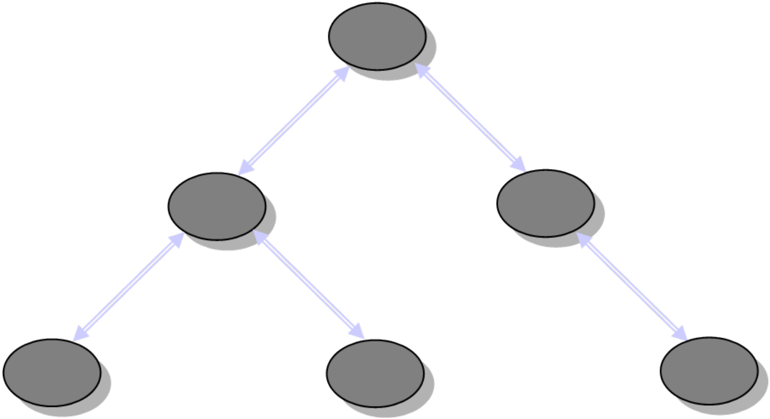
Certificates and PKI

Such a system, together with many details

- How to decide whether to trust Charlie as a CA
- How should Charlie check Bob
- ...

is called a *public-key infrastructure (PKI)*

Hierarchic PKI



Hierarchic PKI

Some specific actors are recognized as CAs[2]

- Your company's security officer
- Professional actors (e.g. Verisign, Thawte, ...)
- Governmental agency (e.g. Belgian eID)
- ...

[2] Which does not mean that everybody trusts them

Hierarchic PKI

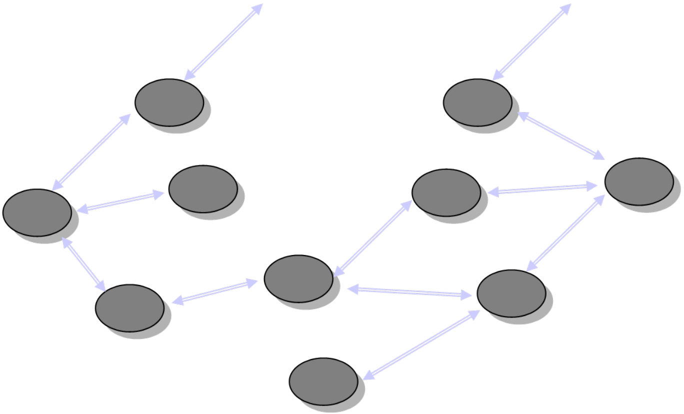
A CA often develops an internal hierarchy

- High-level (super-protected) root CA
- Mid-level CAs, certified by higher levels

Root CA's key must be transmitted in a secure way

- Hand-to-hand given when entering a company
- Embedded in web browser
- Embedded in operating system
- ...

Flat PKI



Flat PKI

No hierarchic view

I trust someone's key

- Because I met him personally
- Or because I trust the key of someone who trusts him, and whom I trust
- (Recursive relationship)

Trust level can be associated to each contact (possible to use combined trust...)

First-hand certificates must be distributed in a secure way

- Face-to-face meeting (e.g. key-signing parties)
- Read over the phone (key fingerprint)
- ...

Typical example: PGP

In practice

Multiple CAs are often used

- Several root CAs embedded in web browsers
- PGP keys signed by multiple contacts

Warning: the security level is that of the weakest CA

- How many root CAs embedded in your web browser?
- In a high security application, all unnecessary ones should be removed

Invalidating certificates

What happens if a secret key is compromised, or if its owner should not be trusted any more?

- The corresponding public key should not be used any more
- But how do we transmit this information to those who have a copy of pk?

Two solutions

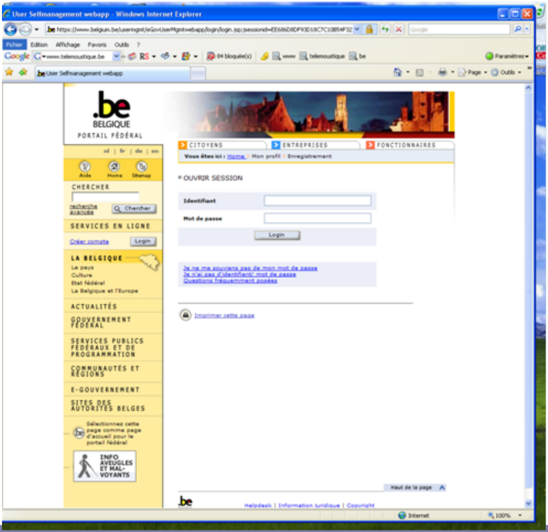
- Limited lifetime (expiration date): ok, but might take long
- Explicit revocation: a message stating "do not trust this key any more"

Certificate revocation list

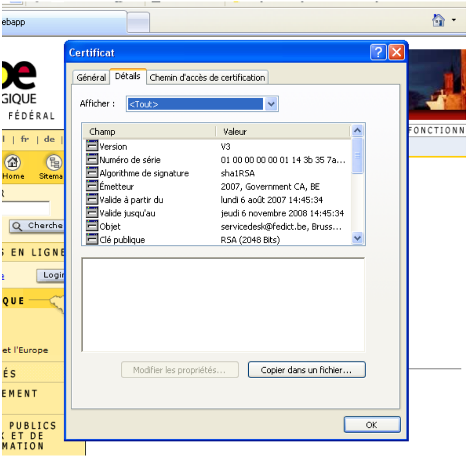
A time stamped list identifying revoked certificates
Signed by a CA (or CRL issuer)
Made available in a public repository
Each revoked certificate is identified in a CRL by its certificate serial number
When a certificate is checked, we must check

- Signatures along certification path
- That the certificate serial number is not on the most recent CRL

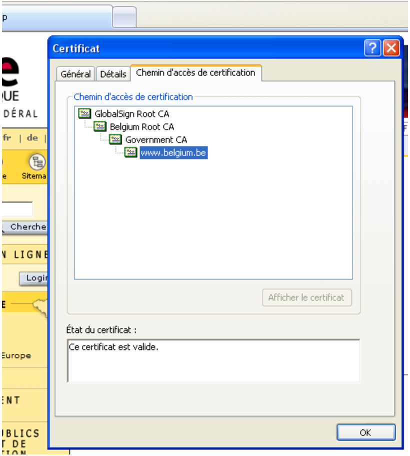
Certificates on the web



Certificates on the web



Certificates on the web



How to register your web server?

Example: Let's Encrypt

Let's Encrypt

`user.org` $\langle pku, sku \rangle$

`letsencrypt.org` $\langle pkl, skl \rangle$

$$\xrightarrow[\text{My public key is } pku]{\text{Hi, I'm } \texttt{user.org}}$$

$$\xleftarrow[\text{And put } \sigma \text{ at } \texttt{user.org/bla}]{\text{Sign } m = \text{"}abcd1234\text{"}}$$

$$\xleftarrow[\text{Here is } \sigma_u = \text{Sign}_{skl}(pku, \texttt{user.org})]{\text{Ok, I checked!}}$$

`user.org` can now advertise pku, σ_u,
which can be verified in any browser knowing pkl