Digital Signatures

Good properties of hand-written signatures:

1. Signature is authentic.
2. Signature is unforgeable.
3. Signature is not reusable (it is a part of the document)
4. Signed document is unalterable.
5. Signature cannot be repudiated.

What problems do we want into if we want to achieve all this in digital signatures ?

Signatures Scheme

To sign: use a private signing algorithm
To verify: use a public verification algorithm

In particular:

Alice wants to sign message m. She computes the signature of m (let's call it y) and sends the signed message (m,y) to Bob.

Bob gets (m,y), runs the verification algorithm on it. The algorithm returns "true" iff y is Alice's signature of m.

How can we do this ? see the board :)

Signatures Scheme

Some public-key cryptosystems can be used for digital signatures, for example RSA, Rabin, and ElGamal:

The basic protocol:

1. Alice encrypts the document with her private key.

2. Alice sends the signed document to Bob.

3. Bob decrypts the document with Alice's public key.

RSA Signature Scheme

1. Alice chooses secret odd primes p,q and computes n=pq.
2. Alice chooses e_A with $\gcd(e_A,\Phi(n))=1$.
3. Alice computes $d_A = e_A^{-1} \bmod \Phi(n)$.
4. Alice's signature is $y = m^{d_A} \bmod n$. $m^{d_A} \bmod n$
5. The signed message is (m,y).
6. Bob can verify the signature by calculating $z = y^{e_A} \bmod n$. (The signature is valid iff m=z).

Alice: publishes $n = 5\cdot 7$, $e_A = 11$ ($\phi(n) = 4\cdot 6$, $d_A = \ldots$)

Potential issues:

Alice wants to send signed $m=6$ $(6, 6^{d_A} \bmod n)$

- Eve could $(y_1^{e_A} \bmod n, y_1)$ to Bob. Is this a problem ?

$(20^{e_A} \bmod n, \underset{y_1}{20}) = (m,y)$ – Bob will verify (of course) that $m = y^{e_A} \bmod n$, so he'll conclude that the message is signed by Alice

Eve produces a Alice-signed message but the message is garbage → not really helpful for Eve

RSA Signature Scheme

1. Alice chooses secret odd primes p,q and computes n=pq.
2. Alice chooses e_A with $\gcd(e_A,\Phi(n))=1$.
3. Alice computes $d_A = e_A^{-1} \bmod \Phi(n)$.
4. Alice's signature is $y = m^{d_A} \bmod n$.
5. The signed message is (m,y).
6. Bob can verify the signature by calculating $z = y^{e_A} \bmod n$. (The signature is valid iff m=z).

Potential issues:

- Eve could $(y_1^{e_A} \bmod n, y_1)$ to Bob. Is this a problem ?

  a valid signed message by "Alice" but the message is garbage

- Bob can reuse the signed message. When would this be a problem ? E.g. checks

Attacks on Signature Schemes

Typical types of attacks for cryptosystems:
ciphertext-only, known-plaintext, chosen-plaintext, and chosen-ciphertext.

Typical types of attacks for signature schemes:

- key-only – Eve has access only to the public key (analogous to ciphertext-only)
- known-message – Eve has a message and its signature
- chosen-message – Eve gets Alice to sign a document

Attacks on Signature Schemes

Additionally, Eve can have different goals:

- total break: Eve determines Alice's signing key/function.

  with RSA signature scheme, not able to do this (we think this is computationally infeasible)

- selective forgery: Eve is able (with nonnegligible probability) to create a valid Alice-signature on a message chosen by someone else.

- existential forgery: Eve is able to create a valid signature for at least one new message.

Some Breaks for RSA Signatures

We mentioned Eve sending $\underline{(y^{e_A} \bmod n, y)}$ to Bob.
What type of attack is this ? key only
What goal does it achieve ? existential forgery

} not a problem since the signed message is garbage

If Eve has two signed messages $(m_1, \overbrace{m_1^{d_A} \bmod n}^{y_1})$ and $(m_2, \overbrace{m_2^{d_A} \bmod n}^{y_2})$, then Eve can create a valid signature on $\underbrace{m_1 m_2 \bmod n}_{\text{new } m}$. How ?

signature for m: $y_1 \cdot y_2 \bmod n = m_1^{d_A} \cdot m_2^{d_A} \bmod n$ " $(m_1 \cdot m_2)^{d_A}$

What type of attack is this ? known message
What goal does it achieve ? existential forgery

Eve can also do a selective forgery using a chosen message attack. How ? Eve gets Alice to sign the selected message ☺

Blind Signatures

Bob wants to time-stamp his document by Alice, without revealing its content to Alice.

1. Alice chooses secret odd primes p, q and computes $n = pq$.
2. Alice chooses e with $\gcd(e, \Phi(n)) = 1$.
3. Alice computes $d = e^{-1} \bmod \Phi(n)$.
4. Bob chooses a random integer k (mod n) with $\gcd(k, n) = 1$, and computes $t = k^{e}m \bmod n$, where m is the message.
   random
5. Alice signs t, by computing $\underline{s} = t^d \bmod n$. She sends s to Bob.
   (t, s)
6. Bob computes $\underbrace{k^{-1}s \bmod n}_{=y}$. This is the signed message m^d. Why ?

Bob can verify that s is a signature for t.

Claim: $(m, y) \to y$ is the signature of m

we know: $y \equiv k^{-1} \cdot s \equiv \pmod n$

$\equiv k^{-1} \cdot t^d \equiv k^{-1} \cdot (k^e m)^d \equiv$

$\equiv k^{-1} \cdot k^{ed} \cdot m^d \equiv k^{-1} \cdot k \cdot m^d \equiv m^d$

by Euler

This protocol is good for Bob but not very good for Alice since she does not know what she is signing !

Insecurity of RSA against Chosen-Ciphertext

Let's revisit this attack (see earlier slides).
Given a ciphertext y, we can choose a ciphertext $\hat{y} \neq y$ such that knowledge of the decryption of $\hat{y}$ allows us to decrypt y.

Choose a random x_0, compute $y_0 = x_0^e \bmod n$

Eve: computes $\hat{y} = y \cdot y_0 \bmod n$

gets Bob to decrypt $\hat{y}$ $\rightarrow$ gets $\hat{x}$, need to multiply by x_0^{-1}

For example (connection to blind signatures):

If $\underbrace{\text{Eve gets Alice to sign } \hat{y}}_{\text{chosen ciphertext}}$, then she has signed message $(\hat{y}, \underbrace{\hat{y}^{d_A}}_{=\hat{x}})$

multiply by x_0^{-1} to get x

Moral of the story: don't sign messages with unknown content!

Combining Signatures with Encryption

If Alice wants to both sign and encrypt a message for Bob:

Either:
Alice signs her message, then encrypts the signed message. I.e. Alice sends $e_{Bob}(m, sig_{Alice}(m))$, where e_{Bob} is Bob's (public) encryption function and sig_{Alice} is Alice's (private) signing function.

Or:
Alice encrypts the message, then signs the encrypted message. I.e. Alice sends $(e_{Bob}(m), sig_{Alice}(e_{Bob}(m)))$.

Eve can create her own signature and remove Alice's (even though Eve does not know what she is signing!)

Which way is better ? sign, then encrypt

Hash Functions

Signature schemes: typically only for short messages (for the RSA signature scheme, messages need to be from Z_n).

also, computation is expensive

What to do with longer messages ?

Naïve solution:

cut it into chunks of size $\approx n$ (each chunk $\in Z_n$)

Problems:

1) need to sign each chunk → computationally expensive
→ signature as long as the document
(for each chunk the signature is $\in Z_n$)

2) Eve can delete / rearrange the chunks

Cryptographic Hash Functions

Using a very fast public cryptographic hash function h, we can create a message digest (or hash) of a specified size (e.g. 160 bits is popular).

What does Alice do ?

Alice wants to sign a (long) message m. She will compute $h(m)$ and signs it.

$(m, h(m), sig_{Alice}(h(m)))$

Alice sends $(m, sig_{Alice}(h(m)))$

How does Bob verify the signature ?

Bob calculates $h(m)$, verifies the signature with $h(m)$.

Cryptographic Hash Functions

Other uses of cryptographic hash functions:

- Data integrity

- Time stamping a message while keeping the message secret

Message m, want to time stamp so that nobody can tweak the message after stamping.

Let's compute $h(m)$ (possibly $h(m + \text{time})$ but this is not necessary for)

Signed Hash Attacks

We have to make sure that h satisfies certain properties, so that we don't weaken the security of the signature scheme.

Attack 1:

Eve finds two messages $m_1 \neq m_2$ such that $h(m_1) = h(m_2)$. Eve gives m_1 to Alice, and persuades her to sign $h(m_1)$, obtaining y. Then (m_2, y) is a valid signed message.

To prevent this attack, we require that h is collision resistant (or strongly collision-free), i.e., it is computationally infeasible to find $m_1 \neq m_2$ such that $h(m_1) = h(m_2)$.

property #1

Signed Hash Attacks

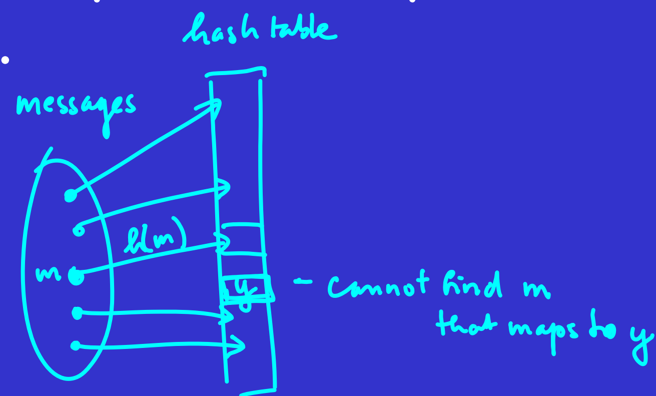
We have to make sure that h satisfies certain properties, so that we don't weaken the security of the signature scheme.

Attack 2: $m_2^{d_A}$

(hashes)

Suppose Eve can forge signatures on random message digests. For example, in RSA, z is the signature of z^{e_A} (m_2). If Eve can find m such that $z^{e_A} = h(m)$, then (m, z) is a valid signed message.

To prevent this attack, we require that h is oneway (a.k.a. preimage resistant), i.e., given y, it is computationally infeasible to find m such that h(m) = y.



Size of Hashes

The birthday paradox:

23 people : about 50% chance of a pair with the same birthday

$$\underbrace{\left(1-\frac{1}{365}\right)}_{\text{prob. of the second person having birthday on a different day than the first}}\cdot\left(1-\frac{2}{365}\right)\left(1-\frac{3}{365}\right)\cdot\ldots\cdot\left(1-\frac{22}{365}\right)\approx 0.5$$

What does it have to do with hashing ?

collisions : with 23 items and hash table of size 365

$\Rightarrow$ 50% chance of a collision

The birthday paradox in general:

$\sqrt{n}$ elements , n - size of the hash table : 50% chance of a collision

Moral of the story: need an appropriate size hash table

Creating Hash Functions

Theoretically appealing option:
creating hash functions from oneway functions, e.g. the Discrete Log (coming soon)

In practice (since the above is too slow):
There are several professional strength hash functions available. E.g., MD4, MD5, and SHA.

DSA (Digital Signature Algorithm)

In 1991, NIST proposed DSA for use in their Digital Signature Standard (DSS). It was adopted in 1994.

There were several criticisms against DSA:
1. DSA cannot be used for encryption or key distribution.
2. DSA was developed by the NSA, and there may be a trapdoor in the algorithm.
3. DSA is slower than RSA.
4. RSA is the de facto standard.
5. The DSA selection process was not public.
6. The key size (512 bits) is too small. In response to this criticism, NIST made the key size variable, from 512 to 1024 bits.

Discrete Log

DSA gets its security from the difficulty of computing the discrete log.

Discrete Log problem:
Fix a prime p. Let α and β be nonnegative integers mod p, the goal is to find the smallest natural number x such that $\beta \equiv \alpha^x$ (mod p). The number x is denoted by $\underline{L_\alpha(\beta)}$: the discrete log of β with respect to α.

Often, α is taken to be a primitive root mod p. α is a primitive root mod p if and only if $\{\underset{\alpha^i}{i} \bmod p \mid 0 \le i \le p-2\} = \{1, 2, \ldots, p-1\}$.

For example:
- 3 is a primitive root mod 7 $\quad 3^0 \equiv 1,\ 3^1 \equiv 3,\ 3^2 \equiv 2,\ 3^3 \equiv 6,\ 3^4 \equiv 4,\ 3^5 \equiv 5,\ 3^6 \equiv 1$
- 2 is a primitive root mod $\underset{= p}{13}$, but $\underset{= \alpha}{3}$ is not

Discrete Log

If α is a primitive root mod p, then $L_\alpha(\beta)$ exists for all $\beta \neq 0$ (mod p).

If α is not a primitive root mod p, then $L_\alpha(\beta)$ may not exist. For example, the equation $\underset{\alpha}{3}^x \equiv \underset{\beta}{2}$ (mod $\underset{p}{13}$) does not have a solution, so $L_3(2)$ does not exist.

α is not a prim. root of 13

e.g. brute force to check that it does not exist

There are Φ(p–1) primitive roots mod p.

Like factoring, the discrete logarithm problem is probably difficult.

Recall: the ElGamal public-key cryptosystem is based on discrete log.