Digital Signatures

Luke Anderson

luke@lukeanderson.com.au

7th April 2017

University Of Sydney

Overview

Digital Signatures

Signatures

Signatures are used to bind an author to a document.

Desirable properties for a signature:

Authentic Sufficient belief that the signer deliberately signed the document.

Unforgeable Proof that only the signer could have signed the document, no-one else.

Non-reusable The signature is intrinsically bound to the document and cannot be moved to another (i.e. be reused).

Unalterable The signature cannot be altered after signing.

Non-repudiation The signer cannot later deny that they did not sign it (most important).

As with all things, these properties can be attacked and subverted. We must consider such attacks when designing systems that use signatures.

Digital Signatures

We have:

m - The message to be signed

k - The secret key

F - The signature scheme (function)

S - The signature

$$S = \mathrm{F}(\mathrm{m}, \mathrm{k})$$

The message m is signed using the secret key k, known only to the signer, which binds the signature S to the message m using some signature scheme F.

Given (m, S) anyone can verify the signature without the secret k.

Non-repudiation is achieved through the secrecy of k.

Digital Signatures with Public Keys

Say Alice wishes to sign a message and send it to Bob

Generation of a key:

1. Alice generates keys:
   - A_v: public (verifying)
   - A_s: private (signing)
2. A_v is published in a public directory
3. A_s is kept secret

Signature Generation:

1. Alice chooses n random bits: $r = \{0,1\}^n$
2. Alice hashes the message to get a message digest: $d = h(m)$
   She uses a collision resistant hash function (CRHF)
3. Alice generates $S = \text{signature}(d, r, A_s)$
4. Alice sends (m, S) to Bob

Signature Verification:

1. Bob obtains A_v from the public directory
2. Bob computes $d = h(m)$
3. Bob runs $\text{verify}(d, A_V, S)$

Attack Models

Total Break
Attacker can recover A_s from A_v and (m, S)

Selective Forgery
Attacker can forge signatures for a particular message or class of message

Existential Forgery
Possible only in theory (based on currently available resources)

https://en.wikipedia.org/wiki/Digital_signature_forgery

Signature Replay

Why might we include $r = \{0,1\}^n$ in the signature?

Consider the following scenario:

- Alice sends Bob a digital cheque for 100.
- Bob takes the cheque to the bank.
- The bank verifies that the signature is valid and credits Bob's account.

What is stopping Bob from cashing the same cheque twice?
(i.e. perform a replay attack)

The random value r is known as a nonce and is used to avoid replay.
(in other words it assures "**freshness**")

The bank keeps track of all nonces it has seen so far from Alice.

Signature based on RSA

A naïve protocol based on RSA might be as follows.

Key Generation:

- $n = pq$
  p, q *are large primes*
- $de \equiv 1 \bmod \phi(n)$
- $A_v = (n, e)$
  public/verifying key
- $A_s = (n, d)$
  private/signature key

Signature Generation:

Assume $m \in Z_n^*$
$S = m^d \bmod n$ — RSA decryption

Signature Verification:

$S^e = m \bmod n$ — RSA encryption

Problems with Naïve RSA scheme

Eve can trick Alice into signing any message m.

Based on RSA's homomorphic property:
If: $s_1 = m_1^d \pmod{n}$ and $s_2 = m_2^d \pmod{n}$
Then: $s_1 s_2 = (m_1 m_2)^d \bmod n$

Attack on naïve RSA scheme:

1. Eve wants Alice to sign hidden message m
2. Eve picks random $r \in Z_n^*$
3. Eve computes $m' = m \cdot r^e \pmod{n}$
4. Eve asks Alice to sign m'
5. Alice returns $s' = (m')^d \pmod{n}$
6. Eve computes $s = \frac{s'}{r} \pmod{n}$

The pair (m, s) is a valid message signature pair!
Eve tricked Alice into signing hidden message m

Note that this trick also works with RSA decryption
(Eve can get Alice to decrypt messages if Alice is not careful)

PKCS#1

PKCS#1 Signature Scheme (RFC2313)

Public Key Cryptography Standards #1

Where the naïve RSA signature scheme has message recovery, the verification function actually returns the message.

PKCS#1 processes a hash instead (much faster)

Signature Generation:

1. $\mathrm{n} = \mathrm{pq}$ (1024-bit modulus)
2. Alice calculates $\mathrm{d} = \mathrm{h}(\mathrm{m})$ (160-bit hash)
3. Define encryption block:
   `EB = [ 00 | BT | PS | 00 | D ]`
   - `PS`: The header is essentially padding
   - `BT`: Block type dictates padding style
   - `EB` is 864 bits + 160 bits = 1024 bits
4. Alice calculates $S = \mathtt{EB}^{\mathrm{d}} (\mathrm{mod} \mathrm{n})$
5. Alice sends (S, m)

PKCS#1 Signature Scheme (RFC2313)

Signature Verification:

- $S = \mathtt{EB}^{\mathrm{d}} (\mathrm{mod} \mathrm{n})$
- Alice calculates $S^e \bmod \mathrm{n} = \mathtt{EB} \pmod{\mathrm{n}}$
- Alice checks the first 864 bits are valid
- Alice checks the last 160 bits are valid (i.e. $= \mathrm{h}(\mathrm{m})$)

Project Tip

You should use a variation of PKCS#1 in part 2 of the project.

ElGamal

ElGamal Signature Scheme

ElGamal is an alternative signature scheme, whose security is based on the **discrete log** problem.

Overview:
Let:

H: a *collision-resistant* hash function

p: a large prime number

g: a randomly chosen integer $< \mathrm{p}$, from the group: $\mathbb{Z}_{\mathrm{n}}^{\times}$

ElGamal Signature Scheme

Key Generation:

- Choose large prime p and generator $g \in \mathbb{Z}_n^\times$
- Choose secret key x where $1 < x < p - 2$
- Compute: $y = g^x \pmod p$
  - Public Key: y
  - Private Key: x

Signature Generation:

- Choose a random k where:
  - $1 < k < p - 1$
  - $\gcd(k, p - 1) = 1$
- Compute: $r = g^k \pmod p$
- Compute: $s = (H(m) - xr)k^{-1} \pmod{p - 1}$
- (if $s == 0$, start again)
- (r, s) *is the digital signature of* m

ElGamal Signature Scheme

Signature Verification:

- Verify: $0 < r < p$ and $0 < s < p - 1$
- Check signature:

$$g^{H(m)} = y^r r^s \pmod{p}$$

If everything checks out, the signature is correct.

Note

Signature generation implies:

$$H(m) = xr + sk \pmod{p-1}$$

Hence:

$$\begin{aligned} y^r r^s &= (g^x)^r (g^r)^{r^{-1}(H(m)-xr)} \\ &= (g^x)^r g^{(H(m)-xr)} \\ &= g^{H(m)} \end{aligned}$$

ElGamal Notes

- ElGamal is rarely used in practice.
  - DSA/DSS is more widely used
- If weak generators (g) are chosen, selective forgery is possible.
- k *must* be random for each signature. If the same k is used twice, then the private key (x) can be recovered.

Digital Signature Standard

Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA) was selected by NIST in 1991 as the Digital Signature Standard (DSS).

Parameter Selection:

- Choose a hash function H
  - Originally SHA-1
  - Now SHA-2 is preferred
- Choose key lengths N & L
- Choose a N-bit prime q
- Choose a L-bit prime modulus p such that $p - 1$ is a multiple of q.
- Choose $g \in \mathbb{Z}_n^\times$ of multiplicative order modulo p is q.
  i.e. $g = h^{\frac{p-1}{q}} \pmod p \neq 1$ for some arbitrary h $(1 < h < p - 1)$

Key Generation:

- Secret key x chosen randomly in: $0 < x < q$
- Compute public key: $y = g^x \pmod p$
  - Also provide p, q, and g parameters as part of public key.

Digital Signature Algorithm (DSA)

Signature Generation:

- Pick random $k \in \mathbb{Z}_n^\times$ where $1 < k < q$
  - Must be unique per message
- Calculate $r = (g^k \pmod p) \pmod q$
  - $r \neq 0$
- Calculate $s = k^{-1}(H(m) + xr) \pmod q$
  - $s \neq 0$
- Signature is: (r, s)

Digital Signature Algorithm (DSA)

Signature Verification:

- Verify:
  - $0 < r < q$
  - $0 < s < q$
- Calculate $w = s^{-1} \pmod{q}$
- Calculate $u_1 = H(m) \cdot w \pmod{q}$
- Calculate $u_2 = r \cdot w \pmod{q}$
- Calculate $v = (g^{u_1} \cdot y^{u_2} \pmod{p}) \pmod{q}$
- Signature is valid if $v == r$

Check it!

Take a look at en.wikipedia.org/wiki/Digital_Signature_Algorithm for the working.

Notes on DSA/DSS

Security analysis of ElGamal is very similar to DSA.

DSA is standard for signatures because:

- DSA cannot be used for encryption (ElGamal can)
- Signatures are short (approx. 320 bits)
- Patent issues

Security of DSA is based on the security of subgroups g.

It is not known whether a sub-exponential algorithm exists in the size of the subgroup for discrete log.

DSA signature verification can be sped up (by a factor of 2) by using simultaneous exponentiation.

One-Way Function Signatures

Signatures based on One-Way-Functions

The **Lamport one-time signature scheme** is a digital signature scheme based on one-way hash functions.

Key Generation:
For a n-bit message, generate $2n \times m$ bit numbers:

$$\{x_1^{(0)}, \ldots, x_n^{(0)}\}, \{x_1^{(1)}, \ldots, x_n^{(1)}\} \in \{0,1\}^m$$

- Public key is: $v_i^{(j)} = H(x_i^{(j)})$ for all i, j
- Private key is all of: $x_i^{(j)}$

Signature Generation:
For a message $M = m_1, \ldots, m_n$
Signature is: $s = (x_1^{(m_1)}, \ldots, x_n^{(m_n)})$

i.e. we select block $x_1^{(0)}$ if bit 1 of m is 0, otherwise $x_1^{(1)}$

Signature Verification:
Test that for all i: $H(x_i^{(m_i)}) = v_i^{(m_i)}$

Signatures based on One-Way-Functions

Notes:

- Only the sender knows the values of x that produce the signature.
- The public key is *very long* and must be unique for every message [1].
- The message itself expands by a factor of m (each bit expands to a m-bit block). Since m must be large to reduce the likelihood of attack, the message expansion is considerable.
- Lamport signatures are believed to be quantum-resistant, unlike ElGamal or RSA based schemes.

[1] unless we use Merkle trees