Digital Signature

Deepak Puthal

Email: Deepak.Puthal@uts.edu.au

41900 – Fundamentals of Security

Overview

- **Digital Signatures**
    - Background
    - Basic Operation
    - Attack Models
        - Replay
        - Naïve RSA
- **PKCS#1**
- **ElGamal**
- **Digital Signature Standard**
- **One-Way Function Signatures**
- **SSL/TLS**

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

$$S = F(m, k)$$

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

Digital Signatures with Public Keys

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*

- $de \equiv 1 \bmod \phi(n)$
- $Av = (n, e)$

  public/verifying key

- $As = (n, d)$

  private/signature key

Signature based on RSA

Signature Generation:

Assume $m \in Z_n^*$

S = m^d mod n — RSA decryption

Signature Verification:

S^e = m mod n — RSA encryption

Problems with Naïve RSA scheme

Eve can trick Alice into signing any message m.

Based on RSA's homomorphic property:

If: $s_1 = m^d{}_1 \pmod{n}$ and $s_2 = m^d{}_2 \pmod{n}$

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 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. n = pq (1024-bit modulus)
2. Alice calculates d = h(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 $S = EB^d (\bmod n)$
5. Alice sends (S, m)

PKCS#1 Signature Scheme

Signature Verification:

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

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 < p, from the group: $Z_n^{\times}$

ElGamal Signature Scheme

Key Generation:

- Choose large prime p and generator $g \in Z_n^\times$
- 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:

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

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 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 $g \in Z_n^{\times}$ of multiplicative order modulo p is q.
  i.e. $g = h^{\frac{p-1}{q}}$ (mod p) ≠ 1 for some arbitrary *h (1 < h < p - 1)*

Digital Signature Algorithm (DSA)

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

Signature Generation:

- Pick random $k \in Z_n^{\times}$ 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) (mod 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 $u2 = r \cdot w \pmod q$
- Calculate $v = (g^{u1} \cdot y^{u2} \pmod p) \pmod q$
- Signature is valid if $v == r$

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

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 speed 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:

$$\left\{x_1^{(0)}, \dots, x_n^{(0)}\right\}, \left\{x_1^{(1)}, \dots, x_n^{(1)}\right\} \in \{0, 1\}^m$$

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

Signatures based on One-Way-Functions

Signature Generation:

For a message M = $m_1, \ldots, m_n$

Signature is: $s = \left(x_1^{(m_1)}, \ldots, x_n^{(m_n)}\right)$

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: $H\left(x_i^{(m_i)}\right) = v_i^{(m_i)}$

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.

SSL/TLS

Raw HTTP is Insecure

In stock standard HTTP, everything goes across the line in plaintext.

This leaves connections vulnerable to:

- message tampering
- eavesdropping on connections
- man-in-the-middle attacks

On top of that, there are numerous flaws in the underlying TCP/IP protocols.

Introducing SSL/TLS

Transport Layer Security (TLS) with its predecessor Secure Socket Layer (SSL) are cryptographic protocols for secure communication.

They use:

- Asymmetric cryptography for authentication
- Symmetric encryption for confidentiality of messages
- MACs for integrity

HTTP sits on top of that to produce HTTPS (HTTP Secure). The only information that leaks is the IP address and TCP port you're connecting to, as well as the size of the messages you're sending and receiving.

While previously only considered for "secure operations", now it's suggested to be used everywhere by default.

TLS Protocol

Client → Server: (SSL version, available ciphers, other info)

Server → Client: (SSL version, select cipher, certificate)

Client authenticates the server (messages sent should be signed by cert).

Client (possibly in conjunction to server, depending on cipher) creates the pre-master secret for the session, encrypts using server's cert, sends to the server.

(Optional) Server may authenticate the client using client's certificate.
All messages from this point are encrypted.

Where do the Certificates come from?

How do we decide whether to trust the certificate the server sends?

- A **Certification Authority** (CA) is a trusted third party that issues digital certificates.
- Certificates for CAs are shipped with operating systems and browsers, and other software.
- Each time a server sends a certificate to a browser, the browser will check to see if one of the CAs it trusts has signed the certificate.

Who are these CAs?

- Small number of multinational companies, with a significant barrier to entry.
- Recent player: https://letsencrypt.org.

Issues with Certification Authorities

Do we actually trust them?

- Many have poor security practises, and are willing to co-operate with governments.
- The small number of root CAs allow other CAs to sign on their behalf.
- In 2011, fraudulent certificates obtained from Comodo were used for MITM attacks in Iran — they can man-in-the-middle any website they want.

They sell based on ridiculous "features":

- Cost of 128-bit and 256-bit certificates are commonly different, even though next to no extra work goes into the second.

Saying "this website secured by SSL" gives a false sense of confidence.

- Very easy for scammers to get themselves an SSL certificate for a domain they own.

How do you acquire a certificate?

Domain Validation: The CA determines you own the domain by one of:

- Having you respond to an email sent to admin@, postmaster@, etc.
- Having you publish a certain DNS TXT record.
- ... and more.

Issue with domain validation: do any of the above methods actually prove that you own the domain?

Extended validation certificates

- CA verifies you pass a set of identity verification criteria.
- Costs more than a normal certificate.
- Puts a green bar at the top of your browser.

Extended validation certificates still don't stop a faked normal certificate from intercepting traffic.

How does it work?

1. Generate a public/private keypair for your server.
2. Generate a Certificate Signing Request (CSR), containing domain name, public key, and other relevant details, and send this to a CA.
3. CA confirms any necessary details (domain validation or extended validation).
4. The CA signs the certificate and sends it back.
5. Install the certificate on your server.

Version: 3
Issuer: GlobalSign
Validity 2015-12-11 to 2016-12-11
Common Name *.wikipedia.org
Public Key Elliptic: 04:cb:· · · :0b:fd
Signature algorithm PKCS#1 SHA-256 RSA
Signature b2:c6:· · · :39:fd

An abridged copy of Wikipedia's current certificate. The last two fields are added by the CA.

Certificate Revocation

There is a method to revoke SSL certificates, called **Online Certificate Status Protocol** (OCSP).

- Whenever a browser sees a new SSL cert, it makes a request to the OCSP URL embedded in the CA's signing certificate.
- The OCSP server sends a signed response indicating whether the certificate is still valid.
- The signature on the response only covers some of the response data, and does not include the response status. Hence a MITM can send back any response status.

The outcome of this is that attackers can intercept an OCSP request and send a *tryLater* response status. All browsers think this is fine (they can't tell otherwise) and avoid raising an issue.

Even if SSL/TLS were perfect...

There are still many attacks on SSL/TLS, even assuming the transport itself is perfect.

Many take advantage of users and the inability of browsers to recommend the correct course of action when things “aren’t right”.

Recommended viewing: the DEFCON presentation “More tricks for defeating SSL in practice” by Moxie Marlinspike.