Message authentication

-- Reminder on hash functions
-- MAC functions
  hash based
  block cipher based
-- Digital signatures

Hash functions

- a hash function is a function \( H : \{0, 1\}^* \rightarrow \{0, 1\}^n \) that maps arbitrary long messages into a fixed length output

- notation and terminology:
  - \( x \) – (input) message
  - \( y = H(x) \) – hash value, message digest, fingerprint

- typical application:
  - the hash value of a message can serve as a compact representative image of the message (similar to fingerprints)
    - \( H \) is a many-to-one mapping \( \rightarrow \) collisions are unavoidable
    - however, finding collisions are very difficult (practically infeasible)
  - increase the efficiency of digital signatures by signing the hash instead of the message (expensive operation is performed on small data)

- examples:
  - (MD5,) SHA-1, SHA-256
Desired properties of hash functions

- ease of computation
  - given an input $x$, the hash value $H(x)$ of $x$ is easy to compute

- weak collision resistance ($2^{nd}$ preimage resistance)
  - given an input $x$, it is computationally infeasible to find a second input $x'$ such that $H(x') = H(x)$

- strong collision resistance (collision resistance)
  - it is computationally infeasible to find any two distinct inputs $x$ and $x'$ such that $H(x) = H(x')$

- one-way hash function (preimage resistance)
  - given a hash value $y$ (for which no preimage is known), it is computationally infeasible to find any input $x$ such that $H(x) = y$

- collision resistant hash functions are similar to block ciphers in the sense that they can be modeled as a random function

Iterative hash functions

- operation:
  - input is divided into fixed length blocks
  - last block is padded if necessary
  - each input block is processed according to the following scheme

  \[
  CV_0 = IV
  \]

  \[
  CV_L = H(x) = CV_0 \oplus f(CV_{L-1}, x_1) \oplus f(CV_{L-2}, x_2) \oplus \ldots \oplus f(CV_1, x_{L-1}) \oplus f(CV_0, x_L)
  \]

alternative illustration:
MAC functions

- MAC = Message Authentication Code
- a MAC function is a function $MAC : \{0, 1\}^* \times \{0, 1\}^k \rightarrow \{0, 1\}^n$ that maps an arbitrary long message and a key into a fixed length output
  - can be viewed as a hash function with an additional input (the key)

- terminology and usage:
  - the sender computes the MAC value $M = MAC(m, K)$, where $m$ is the message, and $K$ is the MAC key
  - the sender attaches $M$ to $m$, and sends them to the receiver
  - the receiver receives $(m', M')$
  - the receiver computes $M'' = MAC(m', K)$ and compares it to $M'$; if they are the same, then the message is accepted, otherwise rejected

- services:
  - **message authentication and integrity protection**: after successful verification of the MAC value, the receiver is assured that the message has been generated by the sender and it has not been altered

- examples:
  - HMAC, CBC-MAC schemes

MAC generation and verification illustrated

[Diagram showing the process of MAC generation and verification.]
**Desired properties of MAC functions**

- **ease of computation**
- **key non-recovery**
  - it is computationally infeasible to recover the secret key $K$, given one or more message-MAC pairs $(m_i, M_i)$ for that $K$
- **computation resistance**
  - given zero or more message-MAC pairs $(m_i, M_i)$, it is computationally infeasible to find a valid message-MAC pair $(m, M)$ such that $m \neq m_i$
  - computation resistance implies key non-recovery but the reverse is not true in general

---

**Secret prefix method**

$\text{MAC}_k(x) = H(k|x)$

- insecure!
  - assume an attacker knows the MAC on $x$: $M = H(k|x)$
  - he can produce the MAC on $x'|y$ as $M' = f(M,y)$, where $x'$ is $x$ with padding and $f$ is the compression function of $H$

---

![Diagram of Secret Prefix Method](image)
A similar mistake

MAC_k(x) = H_k(x)
where H_k(.) is H(.) with CV_0 = k

Secret suffix method

MAC_k(x) = H(x|k)

- insecure if H is not collision resistant
  - using a birthday attack, the attacker finds two inputs x and x' such that H(x) = H(x') (can be done off-line without the knowledge of k)
  - then obtaining the MAC M on one of the inputs, say x, allows the attacker to forge a text-MAC pair (x', M)
- weaknesses
  - MAC depends only on the last chaining variable
  - key is involved only in the last step
**HMAC**

\[ \text{HMAC}_k(x) = H((k^* \oplus \text{opad}) | H((k^* \oplus \text{ipad}) | x)) \]

where
- \( h \) is a hash function with input block size \( b \) and output size \( n \)
- \( k^* \) is \( k \) padded with 0s to obtain a length of \( b \) bits
- \( \text{ipad} \) is 00110110 repeated \( b/8 \) times
- \( \text{opad} \) is 01011100 repeated \( b/8 \) times

Protocols (message authentication, session key establishment)

**CBC-MAC**

- CBC MAC is secure for messages of a fixed number of blocks
- Forgery is possible if variable length messages are allowed
How to use CBC-MAC in practice?

- use the optional final encryption
  - reduces the threat of exhaustive key search (key is \(k, k'\) → key length is doubled)
  - prevents known existential forgeries
  - has marginal overhead (only last block is encrypted multiple times)

- prepend the message with a block containing the length of the message before the MAC computation

- use \(k\) to encrypt the length and obtain \(k' = E_k(\text{length})\), and use \(k'\) as the MAC key (i.e., use message dependent MAC keys)

Digital signature schemes

- functions (algorithms) and terminology:
  - key-pair generation function \(G() = (K^+, K^-)\)
    \(K^+\) – public key
    \(K^-\) – private key
  - signature generation function \(S(K^-, m) = s\)
    \(m\) – message
    \(s\) – signature
  - signature verification function \(V: V(K^+, m, s) = \text{accept or reject}\)

- services:
  - message authentication and integrity protection: after successful verification of the signature, the receiver is assured that the message has been generated by the sender and it has not been altered
  - non-repudiation of origin: the receiver can prove this to a third party (hence the sender cannot repudiate)

- examples: RSA, DSA, ECDSA (shorter key and signature length!)
“Hash-and-sign” paradigm

- public/private key operations are slow
- increase efficiency by signing the hash of the message instead of the message
- it is essential that the hash function is collision resistant (why?)

Security of digital signature schemes

- as in the case of public-key encryption, security is usually related to the difficulty of solving the underlying hard problems

- attack objectives:
  - existential forgery
    - attacker is able to compute a valid signature for at least one message
  - selective forgery
    - attacker is able to compute valid signatures for a particular class of messages
  - total break
    - the attacker is able to forge signatures for all messages or he can deduce the private key

- attack models:
  - key-only attack
  - known-message attack
  - (adaptive) chosen-message attack
**RSA signature scheme**

- **key pair generation**
  - same as for RSA encryption: public key is \((n, e)\), private key is \(d\)

- **signature generation** (input: \(m, d\); output: \(\sigma\))
  - compute \(\mu = h(m)\)
  - (PKCS #1 formatting)
  - compute \(\sigma = \mu^d \mod n\)

- **signature verification** (input: \(m, \sigma, (n, e)\); output: yes/no)
  - compute \(\mu' = \sigma^e \mod n\)
  - (PKCS #1 processing, reject if \(\mu'\) is not well formatted)
  - compute \(\mu = h(m)\)
  - compare \(\mu\) and \(\mu'\)
    - if they match, then output yes (accept)
    - otherwise, output no (reject)

**Management requirements for key pairs**

- RSA has the interesting property that the same key pair can be used for both encryption and digital signature

- however, such double use of key-pairs is not advisable; users should have different key-pairs for different applications

- the main reason is in the difference in key management requirements
  - digital signature
    - private key should never leave the key owner's system
    - private key doesn't need back up and archive (why?)
    - public key (certificate) needs to be archived
  - encryption
    - private key often needs to be backed up and archived (why?)
    - public key usually doesn't need to be archived

⇒ the two applications have conflicting requirements
Session key establishment protocols

-- Motivations and design objectives
-- Basic concepts and techniques
-- Key transport and key agreement protocols
-- Password based key exchange

Motivation

- communicating parties must share a secret key in order to use symmetric key cryptographic algorithms (e.g., block ciphers, stream ciphers, and MAC functions)

- it is desired that a different shared key is established for each communication session \(\rightarrow\) session key
  - to ensure independence across sessions
  - to avoid long-term storage of a large number of shared keys
  - to limit the number of ciphertexts available for cryptanalysis

- we need mechanisms that allow two (or more) remote parties to set up a shared secret in a dynamic (on-demand) manner \(\rightarrow\) session key establishment protocols
Design objectives

at the end of the protocol

- Alice and Bob should learn the value of the session key $K$ (effectiveness)

- no other parties (with the possible exception of a trusted third party) should know the value of $K$ (implicit key authentication)

- Alice and Bob should believe that $K$ is freshly generated (key freshness)

- optionally, Alice should believe that Bob knows the key $K$, and vice versa (key confirmation)

Adversary model

- the underlying cryptographic primitives used in the protocol are secure

- however, the adversary may obtain old session keys

- the adversary has full control over the communications of the honest parties
  - can eavesdrop, modify, delete, inject, and replay messages
  - can coerce honest parties to engage into protocol runs

- the adversary may be a legitimate protocol participant (an insider), or an external party (an outsider), or the combination of both
Basic classification of protocols

- **key transport protocols**
  - one party (typically a trusted third party) creates a new session key, and securely transfers it to the other parties

- **key agreement protocols**
  - the session key is derived by the parties as a function of information contributed by each, such that no party can predetermine the resulting value of the key

First attempt for a key transport protocol

Alice  A, B  Server  generate K  Bob

K  A, K

most obvious problem:
- the adversary can eavesdrop K
- implicit key authentication is not provided
**Second attempt**

**Problems:**
- Alice cannot be sure that K has been created for the session between herself and Bob
- Similarly, Bob cannot be sure that he shares K with Alice
- Implicit key authentication is still not provided
- ...

---

**An attack against the second attempt**

**Notes:**
- Typical man-in-the-middle (MitM) attack
- Alice believes that she shares K with Bob, but she shares it with the adversary

**Derived design principle:**
- *If the name of a party is essential to the meaning of a message, then it must be mentioned explicitly in the message*
Third attempt

### Problem:
- Neither Alice nor Bob can be sure that $K$ is fresh
- No key freshness is provided

---

An attack against the third attempt

### Notes:
- Typical *replay attack*
- If $K$ is compromised by the adversary, then she can decrypt follow-up communications between Alice and Bob
- Even if $K$ is not compromised, the adversary can replay encrypted messages to Alice and Bob from the past session where $K$ was used
How to achieve freshness?

- use timestamps
- use random nonces (nonce = number used once)
- use a key agreement protocol

Timestamps

- $E_{K_{as}}(B \mid K \mid T_s)$, where $T_s$ is the current time on the clock of S
- key is accepted only if the timestamp is within an acceptable window of the current time at the receiver
- can provide strong assurances, but requires synchronized clocks
- important warning:
  if a party’s clock is advanced, then (s)he may generate messages that will be considered fresh in the future (although they may be dropped near the time of their generation)
Random nonces

- $E_{K_{as}}(B \mid K \mid N_A)$, where $N_A$ is a fresh and *unpredictable* random number generated by A (and sent to S beforehand)

- key is accepted only if the time that elapsed between sending the nonce and receiving the message containing the nonce is acceptably short

- less precise than a timestamp (exact time of key generation is not known), but it provides sufficient guarantees of freshness in most practical cases

- it requires an extra message to send the nonce, and some temporary state to store the nonce for verification purposes

**Important warning:**
if nonces were predictable, the adversary could obtain a message containing a future nonce of Alice, which would later be considered as fresh by Alice

Alice believes that the key is younger than $T_A - t$, while in fact, it is older than $T_A - T$
Key freshness in key agreement protocols

- $K = f(k_A, k_B)$, where $k_A$ and $k_B$ are the contributions of Alice and Bob, respectively

- if $f(x, .)$ is a one-way function (for any $x$), then once Alice has chosen $k_A$, Bob cannot find any $k_B$, such that $f(k_A, k_B)$ has a pre-specified value (e.g., an old session key)

- similarly, if $f(., y)$ is a one-way function (for any $y$), then once Bob has chosen $k_B$, Alice cannot find any $k_A$, such that $f(k_A, k_B)$ has a pre-specified value

→ if the contribution of a party is fresh, then (s)he can be sure that the resulting session key is fresh too

Fourth attempt

Alice       Server       Bob

\[ E_{K_{\text{server}}}(B \mid K \mid N_A \mid E_{K_{\text{bob}}}(A \mid K)) \]

\[ E_{K_{\text{bob}}}(A \mid K) \]

\[ N_B \]

\[ E_{K_{\text{client}}}(A \mid B \mid K \mid N_B) \]

notes:
- nested encryption provides key confirmation for Bob
- this protocol is similar to the well-known Needham-Schroeder protocol (symmetric key)
- seemingly correct, but …
An attack against the fourth attempt

\[ E_{K_{bs}}(A|K) \quad N_B' \quad E_{A}(A | B | K | N_B') \]

notes:
- K is an old session key that is compromised by the adversary
- \( E_{K_{bs}}(A|K) \) is replayed from the old protocol run (where K was established as the session key)
- Bob will believe that he established a session with A, but A is not present

derived design principles:
- the fact that a key K is used recently to encrypt a message does not mean that K is fresh
- when proving the freshness of a key K by binding it to some fresh data (timestamp or nonce), don’t use K itself for the binding

Fifth attempt

\[ E_{K_{as}}(B | K | N_A) \quad A, N_A, B, N_B \quad E_{K_{bs}}(A | K | N_B) \quad E_{K_{as}}(B | K | N_A) \]

- any problems?
Protocol engineering checklist

- **be explicit**
  - interpretation of messages shouldn’t depend on context information, but it should be based solely on the content of the messages
  - include **names** that are needed to correctly interpret the message
  - consider including protocol type, run identifier, and message number to avoid protocol interference, interleaving, and message reflection attacks, respectively

- **think twice about key freshness**
  - decide on how you want to ensure key freshness for the different participants
  - consider the advantages and disadvantages of nonces and timestamps in a given application environment

- **state assumptions**
  - explicitly state all the assumptions on which the security of your protocol depends so that someone who wants to use your protocol can verify if they hold in a given application environment

---

Key agreement with the Diffie-Hellman protocol

**summary**: a key agreement protocol based on one-way functions; in particular, security of the protocol is based on the hardness of the discrete logarithm problem and that of the Diffie-Hellman problem

**assumptions**: $p$ is a large prime, $g$ is a generator of $\mathbb{Z}_p^*$, both are publicly known system parameters

**characteristics**: NO AUTHENTICATION, key freshness with randomly selected exponents, no party can control the key, no need for a trusted third party
The Station-to-Station protocol

**summary:** three-pass variation of the basic Diffie-Hellman protocol; it uses digital signatures to provide mutual entity authentication and mutual explicit key authentication.

**characteristics:** mutual entity authentication, mutual explicit key authentication, key freshness with random exponents, no party can control the key, off-line third party for issuing public key certificates may be required, initial exchange of public keys between the parties may be required.

Password based key exchange

- assume that two parties (e.g., a user and a server) share a password (relatively weak secret)
- how to set up a cryptographic key (strong secret) with the help of this password?
A naïve solution

- Alice can generate a key K and encrypt it with the password pwd (or its hash value):

\[ A \rightarrow B : A, E_{H(pwd)}(K) \]

- Bob can use the hash of the password to obtain K from \( E_{H(pwd)}(K) \), and then use K to encrypt messages for Alice for example:

\[ B \rightarrow A : E_K(\text{“Last login at 16:34, Monday”}) \]

The problem

- (key freshness is not provided by the naïve protocol, but it could be added by including a timestamp)

- if a weak password is used, then the naïve solution is vulnerable to an off-line dictionary attack:
  - assume that the attacker eavesdropped a protocol run
  - for each candidate password pwd\( ? \), compute the candidate key 
    \[ K? = D_{H(pwd\( ? \))}(E_{H(pwd\( ? \)})(K)) \]
  - test \( K? \) by checking if \( D_K(E_K(\text{“Last login …”})) \) is a meaningful message
  - if so, then pwd\( ? \) is Alice’s password, otherwise throw away pwd\( ? \) and try a new candidate password from the dictionary
**Encrypted Key Exchange (EKE) – the basic idea**

- Alice generates a public key / private key pair $K^+$ and $K^-$, and encrypts $K^+$ with the (hash of the) password pwd:

  $$A \to B : A, E_{H(pwd)}(K^+)$$

- Bob uses the (hash of the) password to obtain $K^+$, then generates a (symmetric) key $K$, and encrypts it with $K^+$ in the public key cryptosystem; the result is further encrypted with the (hash of the) password:

  $$B \to A : E_{H(pwd)}(AE_{K^+}(K))$$

- Alice uses the (hash of the) password and $K^-$ to obtain $K$ from $E_{H(pwd)}(AE_{K^+}(K))$; then she can use $K$ to send messages to Bob:

  $$A \to B : E_K(\text{"Last login at 16:34, Monday"})$$

**Why is this good?**

- for a candidate password $pwd^?$, the attacker can compute a candidate public key $K^+^?$ as $D_{H(pwd^?)}(E_{H(pwd)}(K^+))$

- but $K^+^?$ cannot really be tested
  - the attacker needs to find a key $K^?$ such that
    - $AE_{K^+^?}(K^?) = D_{H(pwd^?)}(E_{H(pwd)}(AE_{K^+}(K)))$
    - $D_K(E_K(\text{"Last login …"}))$ makes sense
  - both would require an exhaustive search over the key space from which $K$ is chosen (or breaking the symmetric or the asymmetric cipher)

  → the relatively small space of passwords is thus multiplied by the large key space from which $K$ is chosen (privacy amplification effect)
What about key freshness?

- as Bob generates $K$, key freshness is provided for Bob
- for Alice $K^+$ is fresh, and this guarantees freshness of $K$ through the encryption $AE_{K^+}(K)$ (assuming that Alice trusts Bob for generating fresh session keys)
  - Alice can conclude that someone who knows the password (which can only be Bob) has recently sent $K$ to the other holder of the password (which can only be Alice)