Security in Computer Networks

Goal: we want to enable secure communication between two parties in a distributed system. This requires implementing three related **security functions**:

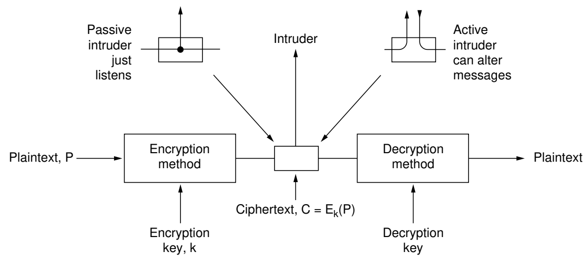
Authentication: (1) Ensuring that a message is genuine, has arrived exactly as it was sent, and came from the stated source; (2) Verifying the identity of an individual, such as a person at a remote terminal or the sender of a message.

Data integrity: The property that data has not been altered or destroyed in an unauthorized manner.

Confidentiality (secrecy): The property that information is not been made available or disclosed to unauthorized individuals, entities, or processes.

Implementing these functions generally requires the use of **cryptographic protocols**.

Cryptography



Cryptographic functions:

Secret key: Use a single key to (1) encrypt the plaintext and (2) decrypt the ciphertext. Requires that sender and receiver **share** the secret key.

Public key: Use different keys for encryption and decryption, of which one is **private**, and the other **public**.

Hashing: Just use a hash function on the plaintext and send it off. There's no decryption at all, just verification.

Hash algorithms

Idea: take an arbitrary length message, and compute a unique, fixed-length number from it. Also called **message digest**, or **one-way function**.

- Computing the hash $h(m)$ for any message m is relatively easy.
- Given a hash value $h(m)$, the only way of getting m is to enumerate over all possible messages. In other words, h^{-1} is almost impossible to find.
- It is computationally infeasible to find two messages m_1 and m_2 such that $h(m_1) = h(m_2)$.

Used for: password hashing (store hash values for comparison instead of cleartext passwords), message fingerprinting (add a message digest to the message to safeguard against changes), signatures (sign the message digest instead of the entire message).

Authentication

Note: The whole business of security is that we can ensure authorized access to resources. In practice, this means that we pay a lot of attention to authentication first. Confidentiality, i.e. privacy, is practically less important.

Question: this is not entirely true – what's the big exception here?

Note: A stronger version of authentication is **nonrepudiation**: it is not possible for someone to **deny** that they sent a message.

Question: How can we safeguard against repudiation?

Authentication versus Integrity

Note: Authentication and data integrity rely on each other: Consider an active attack by an enemy X on the communication from A to B.

Authentication without integrity: A's message is authenticated, and intercepted by X, who tampers with its content, but leaves the authentication part as is. B will conclude the message came from A – it came from X, so authentication fails.

Integrity without authentication: X intercepts a message from A, and then makes B believe that the content was really sent by X. The data has now been "changed" in an unauthorized manner, so integrity is violated. In other words: integrity is meaningless if you don't know the source of information.

Question: What can we say about confidentiality versus authentication and integrity?

Maintaining integrity I

Idea: if we can't ensure integrity *during* a session, authentication becomes meaningless. Therefore, we have to use cryptography, and thus keys.

Problem: we have to come up with an appropriate key for data exchange. Take the following into account:

Keys wear out: The more data is encrypted by a single key, the easier it becomes to find that key. *Don't use keys too often.*

Danger of replay: Using the same key for different communication sessions, permits old messages to be inserted in the current session. *Don't use keys for different sessions.*

Compromised keys: If a key is compromised, you can never use it again. Really bad if *all* communication between Alice and Bob is based on the same key over and over again. *Don't use the same key for different things.*

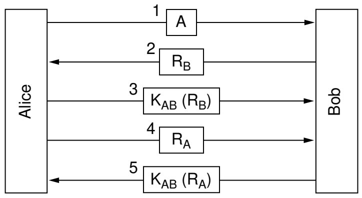
Maintaining integrity II

Temporary keys: Untrusted components may play along perhaps just once, but you would never want them to have knowledge about your really good key for all times. *Make keys disposable.*

Essence: Don't use your valuable and expensive public & private key combination for all your communication, but just use it for authentication purposes. The same goes for certain shared secret keys.

Solution: introduce a **session key** that is used only once, namely during a single conversation/connection ⇒ construct a **secure channel**.

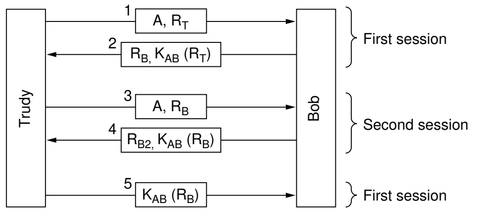
Authentication Protocols Secret Keys



1. Alice sends ID to Bob
2. Bob sends challenge R_B (i.e. a random number) to Alice
3. Alice encrypts R_B with shared key K_{AB}. Now Bob knows he's talking to Alice
4. Alice send challenge R_A to Bob
5. Bob encrypts R_A with K_{AB}. Now Alice knows she's talking to Bob

Note: We can "improve" the protocol by combining steps 1&4, and 2&3. This costs only the correctness.

Authentication Protocols Secret Keys – Reflection Attack



1. Trudy claims she is Alice, and sends challenge R_T
2. Bob sends back a challenge R_B and the encrypted R_T
3. Trudy starts a second session, claiming she is Alice, but uses challenge R_B
4. Bob sends back a challenge, plus $\{R_B\}_{K_{AB}}$.
5. Trudy sends back $\{R_B\}_{K_{AB}}$ for the first session to prove she is Alice

Key management

Idea: If authentication is based on cryptographic protocols, and we need session keys to establish secure channels, who's responsible for handing out keys?

Secret keys: Alice and Bob will have to get a shared key. They can invent their own and use it for data exchange. Alternatively, they can trust a **key distribution center** (KDC) and ask it for a key.

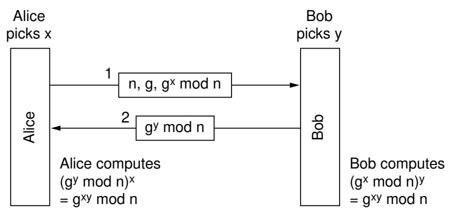
The KDC hands out a **ticket**: an encrypted message containing the shared key.

Public keys: Alice will need Bob's public key in order to decrypt (signed) messages from Bob, or to send private messages to Bob. But she'll have to be sure about actually having Bob's public key, or she may be in big trouble (why?). Use a trusted **certification authority** (CA) to hand out public keys.

The CA hands out a **certificate**: a signed message specifying a user's name and the user's public key.

Establishing a Shared Key: Diffie-Hellman

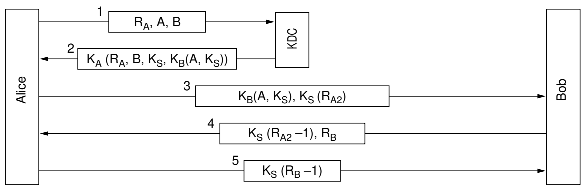
- Alice and Bob have to agree on two large prime numbers, n and g. Both numbers may be public.
- Alice chooses large number x, and keeps it to herself. Bob does the same, say y.



1. Alice sends $(n, g, g^x \bmod n)$ to Bob
2. Bob sends $(g^y \bmod n)$ to Alice
3. Alice computes $K_{AB} = (g^y \bmod n)^x = g^{xy} \bmod n$
4. Bob computes $K_{AB} = (g^x \bmod n)^y = g^{xy} \bmod n$

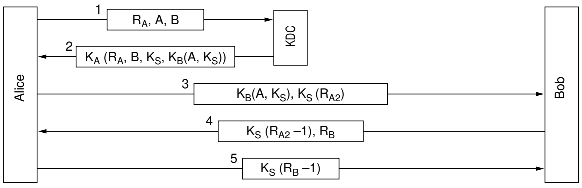
Needham-Schroeder (simplified)

Idea: There is a KDC which shares a key with a number of **principals**. A principal can request a session key to be used for a secure channel with another principal (also known to the KDC).



1. Alice asks KDC a session key for channel to Bob, with challenge R_{Alice}. KDC generates K_{AB} and **ticket** $T_{AB} = \{\mathsf{Alice}, K_{AB}\}_{K_{Bob}}$.
2. KDC sends $\{R_{Alice}, \mathsf{Bob}, K_{AB}, T_{AB}\}_{K_{Alice}}$.

Needham-Schroeder (cont'd)



3. Alice just sends the ticket, as well as a challenge $\{R^*_{Alice}\}_{K_{AB}}$. Bob retrieves the session key from the ticket.
4. Bob sends proof back and challenges Alice: $\{R^*_{Alice} - 1\}_{K_{AB}}, R_{Bob}$.
5. Alice returns proof of the challenge: $\{R_{Bob} - 1\}_{K_{AB}}$.

Needham-Schroeder (cont'd)

Q1: Why does the KDC put `Bob` into its reply message, and `Alice` into the ticket?

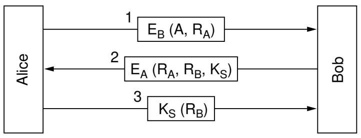
Q2: The ticket sent back to Alice by the KDC is encrypted with Alice's key. Is this necessary?

Security flaw: Suppose Trudy finds out Alice's key ⇒ she can use that key anytime to impersonate Alice, even if Alice changes her private key at the KDC.

Reasoning: Once Trudy finds out Alice's key, she can use it to decrypt a (possibly old) ticket for a session with Bob, and convince Bob to talk to her using the old session key.

Solution: Have Alice get an encrypted number from Bob first, and put that number in the ticket provided by the KDC ⇒ we're now ensuring that every session is known at the KDC.

Authentication Protocols Public Key



1. Alice sends a challenge R_A to Bob, encrypted with Bob's public key E_{Bob}.
2. Bob decrypts the message, proves he's Bob (by sending R_A back), and sends a challenge R_B to Alice, along with a session key K_S. Everything's encrypted with Alice's public key E_{Alice}.
3. Alice proves she's Alice by sending back the decrypted challenge, but now encrypted with the session key K_S.

Digital Signatures

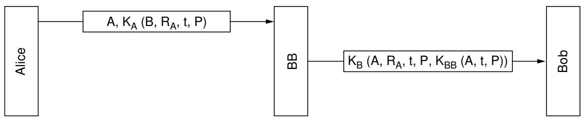
What we often really need is to authenticate the message, and assure its integrity:

1. Receiver can verify the claimed identity of the sender
2. The sender can later not deny that he/she sent the message
3. The receiver can not tamper the message itself.

The solution is to **digitally sign** the message. This means:

- have the sender put a signature that can be verified
- be sure that the signature cannot be faked, i.e. it should be uniquely associated with the message.

Secret-Key Signatures



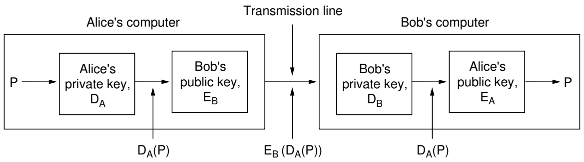
Basic idea: Pretty simple – just use a *Big Brother* who passes the message, but signed, to the destination:

1. Alice sends $\{\text{Bob}, R_{Alice}, t, P\}_{K_{Alice}}$ to Big Brother.
2. Bib Brother signs (Alice, t, P), and sends it along with the original message, encrypted with Bob's secret key: $\{\text{Alice}, R_{Alice}, t, P, \{\text{Alice}, t, P\}_{K_{BB}}\}_{K_{Bob}}$.

Note: Using R_{Alice} and timestamps helps against replays.

Question: Why is signing by Big Brother necessary?

Public Key Signatures



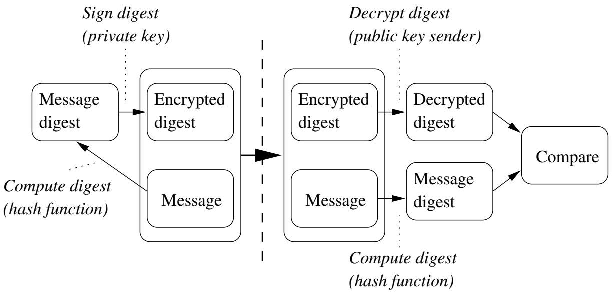
1. Alice encrypts her message P with her private key: $P_{Alice} = \{P\}_{K_{Alice}}$.
2. She then encrypts P_{Alice} with Bob's public key: $\{P_{Alice}\}_{K^*_{Bob}}$, and sends it off.
3. Bob decrypts the incoming message with his private key K_{Bob}. We know for sure that no one else has been able to read P_{Alice} during its transmission.
4. Bob decrypts the message with Alice's public key K^*_{Alice}. Bob now knows the message came from Alice.

Note: we're assuming that $\{\{P\}_{K_A}\}_{K^*_A} = P = \{\{P\}_{K^*_A}\}_{K_A}$.

Message Digests

Basic idea: Don't mix authentication and secrecy. Instead, it should also be possible to send a message in the clear, but have it signed as well.

Solution: take a message digest, and sign that:

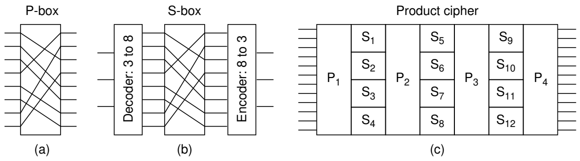


Cryptology – Devising Ciphers

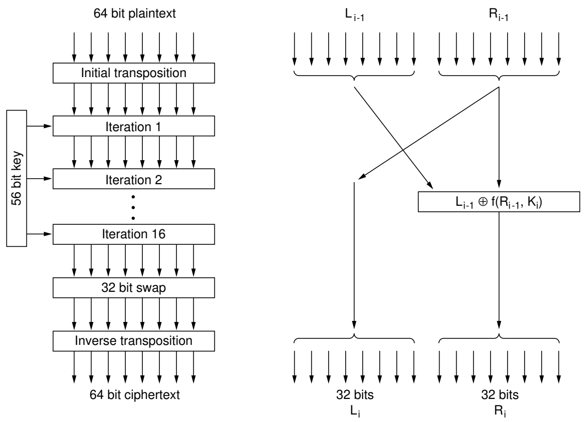
Substitution: Just replace some characters with others following a predefined mapping ⇒ implement by means of an **S-box**.

Transposition: Reshuffle the characters following a predefined pattern (e.g., a transposition table) ⇒ implement by means of an **P-box**.

Combine: Cascading a lot of S and P boxes does the trick.



DES: Data Encryption Standard



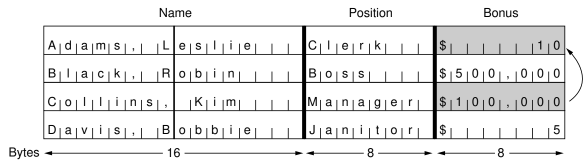
(a) (b)

- Each iteration i uses a different key K_i. The complexity lies in the mangler function f.
- The keys K_i are derived from the initial 56-bit key.

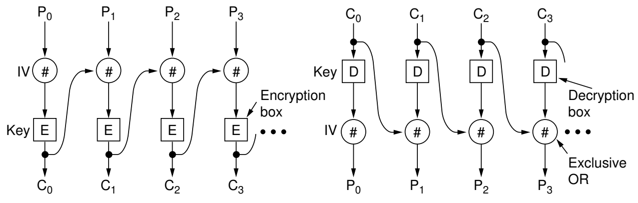
The real problem is the 56-bit key: it's too easy to break.

DES Chaining

Problem: A 64-bit plaintext will always come out the same way. Great, now we can fool around a bit – just see if you can divide your input file into chunks of 64 bits *width*:



Solution: Use a predecessing, encrypted block to permute the current block before it's encrypted. Decryption works the other way around.



(a) (b)

RSA: Rivest, Shamit, Adleman

Really nifty: The whole idea is that you use a private and a public key. First find the right number for encryption:

1. Choose two large primes p and q
2. Compute $n = p \times q$ and $z = (p-1) \times (q-1)$
3. Choose a number d relatively prime to z
4. Find e such that $e \times d = 1 \text{ mod } z$

Next step: Consider your plaintext as a bitstring; divide into blocks, where each block is considered to be a binary number $0 \le P < n$.

Sending: encrypt each message P into C as: $C = P^e(\text{ mod } n)$, and send it off. **Note:** you need e and n.

Receiving: decrypt an incoming message into $Q = C^d(\text{ mod } n)$. Guess what: $Q = P^{e \cdot d} = P$. **Note:** you need d and n.

Note: if $\gcd(a,n) = 1$, then $a^{(p-1)(q-1)} \text{ mod } n = 1$. Consequently, $P^{e \cdot d} \text{ mod } n = P^{e \cdot d - 1} \times P \text{ mod } n = P \text{ mod } n$.