Last time we discussed a scenario where \( n \) users share a single key. They used Encryption and message authentication, but this required that each pair share a key, which isn’t really practical.

## 1 Session Key Exchange

Session Key Distribution (SKD) in a symmetric key setting fixes this problem. The goal of this protocol is to allow any pair to initiate a secure session without having to remember lots of keys. To this end, let each user have an ID (an email address, for example) and let the Key Distribution Center (KDC) be a trusted central authority. In the initialization phase, each user \( uid_i \) registers with the KDC, and the KDC records in a table the secret key \( k_i \) they decided upon. User \( uid_i \) needs only to remember this key, as opposed to before when he needed to remember \( n \) keys.

Now, if a user wants to initiate a conversation with another user, he needs to somehow generate a fresh pseudo-random session key. This key should be independent of other users’ keys, and also independent of keys used for the user’s previous conversations.

Protocols written back in the 70’s to accomplish this lacked proper formalization, so the desired goals of such protocols weren’t clear. This resulted in protocols with undesirable side effects, which we’ll mention below. To get around these problems, we’ll provide our own notions of security as we present the problems with other protocols.

**Vulnerable protocol:** \( i \) wants to talk to \( j \), and he sends to the KDC the message \( \langle uid_i, uid_j \rangle \) requesting the connection to be initiated. KDC then generates key \( k \), sends \( \{uid_j, \{k\}k_i\} \) to \( i \) and \( \{uid_i, \{k\}k_j\} \) to \( j \).

Let’s look at a few problems with this protocol.

**Problem 1:** One problem with this protocol is that an Adversary can replace \( \{uid_j, \{k\}k_i\} \) with \( \{uid_i, \{k\}k_i\} \). (We assume that the Adversary has complete control over the network, so he can drop or invent messages at will.) This would be very bad; if \( j \) is a bank, for example, \( l \)’s bank account could be credited or debited instead of \( i \)’s.

**Quick fix:** Placing \( uid_i \) inside the connection message will effectively solve this problem. KDC sends \( \{\{k,uid_j\}k_i\} \) to \( i \) and \( \{\{k,uid_i\}k_j\} \) to \( j \).

**Problem 2:** *Replay Attack.* Adversary sends previously used \( \{\{k,uid_j\}k_i\} \) to \( i \) and \( \{\{k,uid_i\}k_j\} \) to \( j \).
Fix: The solution of this problem involves introducing nonces, as was done in previous lectures. But this requires us to formalize our goals of security. We’ll do this by comparing an attack game to an ideal game.

Formally, define the attack game as follows: Each user keeps track of how many instances of the protocol he’s running. Each instance will be referred to by a unique instance ID iid, (that is, an ID not previously used by the current user i). There are also corrupted users controlled by the Adversary; the Adversary chooses the corrupted users when they are initiated. The Adversary

- registers users.
- initiates conversation instances, where conversation instance p of user i with user j is [i : iid, j : iidj].
- accepts and delivers protocol messages.

When a user instance terminates successfully, the player gives session key to the adversary. The point of giving the session key to the Adversary (which is generally a remarkably bad idea) will become clear soon.

Now we define the idealized version of the attack game: Let F be a pseudo-random mapping of session ID’s to session keys (where a session ID is of the form [(uid,iid),(uid iid)]). In the ideal game the random key is given to the Adversary instead of the real session key. Security is defined as the Adversary not being able to distinguish between the attack game and the ideal game. That is, he cannot tell if he received the true key or a random key. If we can show that the adversary can’t differentiate at this level of the model, we can use a hybrid argument to demonstrate that higher levels are secure as well.

The secure model is one where the KDC sends to i the pair [k,SID]. The full proof that this is secure is omitted.

2 Public Key Cryptography

Public Key Cryptography (PKC) circumvents the need for KDC. It still requires a secure central authority, but that authority is less involved in the conversations.

The ’76 Diffey-Hellman paper gave incomplete solutions to the following model of a public key encryption (PKE): A KeyGen algorithm generates a public key (PK) and a secret private key (SK). PK is sent to Alice, and SK is sent to Bob. Alice uses PK to encrypt her message, and Bob uses SK to decrypt it. Diffey-Hellman envisioned a telephone book-like registry that stores all public keys, but in practice this didn’t seem to work too well, since there’s an issue in distributing all the keys securely.

2.1 Example of a PKE scheme.

Consider the following definition of a PKE scheme:
- Attack Game: KeyGen generates \{PK, SK\} and discards the secret key. PK is given to the Adversary and Challenger. The adversary generates two messages and sends \{m_0, m_1\} to the Challenger. The Challenger chooses \(b \leftarrow \lambda\), encrypts \(E_{PK}(m_b)\), and sends the encrypted message back to the Adversary. Finally, the Adversary calculates \(\hat{b}\).

- The basic security notion is known as "semantic security," and defined similar to that of message security for symmetric encryption. We require for any probabilistic polytime algorithm \(A\) that \(\text{prob}[b = \hat{b}] - \frac{1}{2} = \epsilon\) where \(\epsilon\) is negligible.

Note that in the PKE model, multi-message security is implied by single message security! This was not the true concerning the asymmetric security model that showed up in the homework. The reason that this is true has to do with the fact that the Adversary can calculate \(E_{PK}\) himself on random bit strings, and so knowing this gives him no additional information.

A simple (but perhaps inefficient) way to encrypt a single bit for the PKE model is to use Trap Door One Way Permutations, for example RSA and squaring. As usual, we need to extract a random bit. To this end, let KeyGen generate a description of the TDOWP along with the Trap Door - the public key is the description, and the trap door is the secret key. The encryption, given the random bit \(r \leftarrow R\ Domain\), is \(c \leftarrow [\alpha, \beta]\), where \(\alpha = \text{Perm}(r)\), and \(\beta = \text{HCB}(r \oplus m)\). The decryption is \(m' \leftarrow [r' \leftarrow \text{Perm}^{-1}(\alpha), m = r' \oplus \beta]\).

### 2.2 Another example.

Digital Signatures (DS) are another application of PKE. Let KeyGen generate PK and SK. The user can use SK to produce a signature on a string of bits. Any other user can use Pk to verify the signature.

DS can be used to implement a “certificate authority,” which is a database that can store and distribute public keys. The user can verify that the information can from the certificate authority, so it believes that the key is in fact correct, and didn’t come from someone imitating the user.