Small exercise: I ask the following, imagine you are downloading an app for your smarthphone from a market like app store or play store (so from a trusted repository), the problem I am focusing is:
The organization who developed the app of course is providing a *backend* (you can imagine the app as a *client-server* application, you have the client and the organization has the server), then the organization wants all users to use the original app, the organization wants to make authentication on the app (not on the users) so it can recognize that the app that the client is using is the original one on the playstore, then if ok I can continue with authenticating the user. I am making some simple remarks:

1. I can accept that the market is trusted
2. The market can generate a pair of public/private key for any download of any app
3. The smarthphone downloads not only the app but also the public key of the server

When the app runs it performs a three way handshaking. The server gets a list of public keys of clients (each time a client connects to the server the app provides its public key).
Of course this scenario is not completly robust because if the attacker is taking the app and is able to do some reverse engineering on the code the attacker may find the private key. Obviously the private key of the server is not stored clearly in the app, it is encrypted by some other key, but if you encrypt some key by some other key the software (our app) should be able to use the encrypted key so you may try the key in the software.
Solution attempt: Server encrypts K_s with K_p and stores in the app the encrypted K_s. When client downloads the app, he downloads even the encrypted K_s. When the client connects to the Server, he sends it the encrypted K_s, then the Server will decrypt it with K_s. If it obtains K_s, then the app is authentic and the server sends back an ACK, otherwise it sends back a NACK. When the client receives the ACK then the conversation can start.
Note that an attacker cannot be able to compute $E_{Kp}(K_s)$ because he does not know K_s.

AUTHENTICATION

Authentication is the procedure of providing the identity to a party in order to access services that this party provides to the allowed ones. For example Alice needs to show her identity to Bob because Bob wants to show informations only to authorized people.
One of the most common attacks against authentication is spoofing, that is the ability to cheat about the *identity* of the originator of the message. With authentication we want to prevent *spoofing attacks*.
This lifts anoter question: what is the *identity*? In particular *Digital Identity*.
Digital identity is a very complex concept. We do not face this topic because of it's complexness, but in few words digital identity is for example credentials in order to read emails. It is associated to some physical person.

"SPID: have a look on it!"

In many cases in human interactions you recognize the face, the voice and so on… On slide 3 there are described 2 interesting cases of authentication:

1. A computer is authenticating another computer (for example I can design an app and put it on *playstore* and design it such that it talks only with other instance of that app, no compatible apps are allowed).
2. A person is using a public workstation (for example in internet cafee).

Closed world assumption: phylosophical assumption, many people don't like it. It is an idea based on the following sentence: 'what is not currently known to be true, is false'. It believes false every predicate that cannot be proved to be true. Many people don't like it because they prefere the concept of *black list* and *white list*:

- Black list = forbidden things, bad things
- White list = allowed things

Closed environment: your environment is considered closed when you are working in a closed place, like your enterprise. Many people should authenticate when they reach the office, maybe they authenticate in the domain of the office. The larger is the enterprise, the more secure is the authentication, because big enterprises know the cybersecurity importance. In many cases you are somewhat dealing with a third-party (a trusted server) that is protected by many methods, this server can support authentication of many users, we will see Kerbersos that is a very secure environment. In this setting it is very common that you have the trusted server, you have Alice and Bob that need to communicate in a secure way. If the attacker belongs to the same company, it is called an *insider*. The insider has a lot of power compared to an external one.

Human vs Computer authentication: there are many differences, a computer can store a very huge information, high quality secrecy, computers easly store long keys. Humans they are not able to do this, they invent some password. What is the difference between a password and a key? First difference is: since humans are stupid they use password that contains words that are meaningful, words belonging to some dictionary or simple variations. Attackers can run what is called *dictionary attack*. Can I use some such password in order to encrypt? Imagine you use **AES** with the password as a key, you can do it but it is a completly insecure approach.

"A more secure approach is: I invent a password, I hash it, and the result is my key."

Authentication of people: there are many approaches (slide 7).

Passwords: humans can afford short passwords, somethimes are easy to be guessed by the attacker. Computers can store them encrypted. One of the first attack is **Trojan Horse** attack (D'Amore did it).

Scenarios

There are three main approaches in order to achieve authentication, and they are:

1. *Shared secret key*: Alice and Bob share a secret and use it to authenticate each other.
2. *Authentication through a Third Party*: there is a server that shares a secret for every user, if two users have to authenticate each other they pass through the server.
3. *Public Key Criptography*: users use the Public Key Infrastructure in order to achieve authentication.

Of course we can tract all of them, but we start with some important settings. We study them in the order they appear.

Techniques: you can define many techniques, you can think of the exam where when I ask you to design an authentication system, solution is never unique, so there can be many solutions to the question the professor poses. A possible ingredient could be a **Timestamp**, another could be a **Nonce** (random number to be used just once). In many cases it is used to do some challenge, so i just generate a random number and i communicate the number to the other party and i expect that other party can use it according to some agreement we decided in advance, because I expect that only one person can do a specific thing when i give him that random number. Another tecnique could be S**equence numbers**: TCP for example uses sequence numbers in order to do some things like maintaining the sliding window and to track the order of connections.

Cryptography vs Authentication: we have discussed it many times, we are aiming at authenticating, we are not aiming to have confidentiality, so if we encrypt, we do not encrypt messages in order to have confidentiality, we encrypt keys in order to implement authentication. If you also need confidentiality you should add an extra tool that encrypts also messages. We are dealing with encryption for AUTHENTICATION purposes, not CONFIDENTIALITY purporses.

Authentication basing on symmetric key

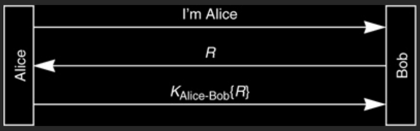
Before starting, some notation:

K_{AB} = key shared by Alice and Bob

K{M} denotes M encrypted with secret key K.

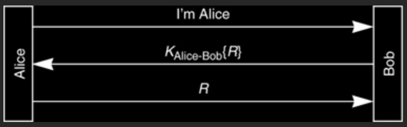
Warning: the **nonce** is R, the **challenge** can be both R or $K_{AB}\{R\}$**.** The meaning is that in order to authenticate Alice, Bob sends her a challenge in order to see if he can trust Alice, because the only way to trust her is that Alice succeedes in winning the challenge. The challenge can be implemented in two ways: either Bob sends a number to Alice and she has to respond with the encrypted one, or Bob sends the encrypted number to Alice and she has to recover the number. In both the cases we are talking about **challenge**, its meaning does not change. So from now on we will use the term challenge referred to a message sent by a party when this party expects that the other party will win the challenge.

Challenge/Response Authentication:



Alice wants to authenticate to Bob. She sends him a plain text presenting herself, Bob responds with a challenge (that is R, the nonce), then Alice encrypts R with the key and sends back to Bob the encrypted R (the correct answer to the challenge). So Bob expects from Alice the right encryption, if he gets the right encryption (the one that cames out using the private key) he authenticates Alice. Of course authentication is not mutual. This scheme can be easly attacked, because Alice cannot be sure that she is authenticating actually to Bob, that Bob could be an attacker who is able to send her a challenge R so when Alice will reply with K{R} the attacker just ignore the cipher and authenticate Alice.

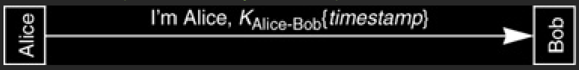
Variant:



In this variant Alice again starts presenting herself, but now the challenge sent by Bob is the encrypted R and then Alice is expected to find the nonce R, then she sends R to Bob in order to be authenticated. This variant provides a little bit more security, because when Bob sends the challenge, we understand that only Bob can generate that challenge (only him knows the private shared key), so Alice can be sure it is Bob that is sending the challenge. In the previous example this was not true. So Alice can trust the party with whom she is authenticating.

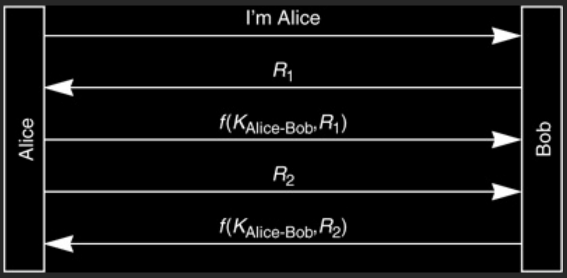
If R has a limited lifetime (R = Random Number + Timestamp), meaning that we can consider it valid for a small amout of time after which it is not valid anymore, then Alice trusts Bob, because the limited lifetime of R does not allow *replay attacks*.

Variant 2 (timestamp based):



Alice sends to Bob the encrypted timestamp (they must have their clock synchronized), then if Bob can decrypt he can authenticate Alice. Of course timestamp has to be a timeout after which it expires, therefore if Bob saves that timestamp until it expires, no replay attack is possible. This solution is similar to the two previous solutions just seen, but it is an optimization because two messages are exchanged (the one displayed, and the ACK from Bob).

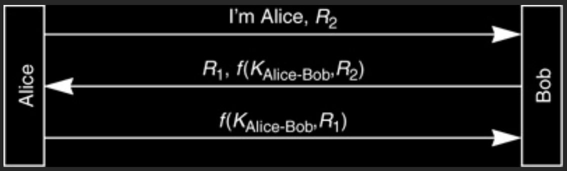
Mutual Authentication



Let's just try twice the one-way authentication. So Alice presents to Bob, Bob sends the nonce R_1, Alice sends back the encrypted nonce $E_k(R_1)$, now Alice is challenging Bob so she sends him the nonce R_2 and Bob sends back the encrypted nonce $E_k(R_2)$.

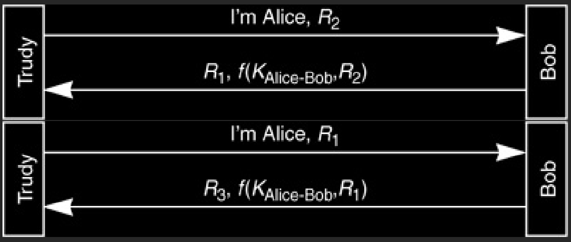
There is a little overhead, can we save some message? Yes!

Optimized variant:



There is a compact way to do mutual authentication: Alice presents to Bob and sends also her challenge R_2, Bob replies with his challenge R_1 and with the response to Alice's challenge. Finally Alice sends back the response to Bob's challenge. In this way we only exchange 3 messages, and the result is the same as before, however, this soffers for the *reflection attack*.

Reflection attack:

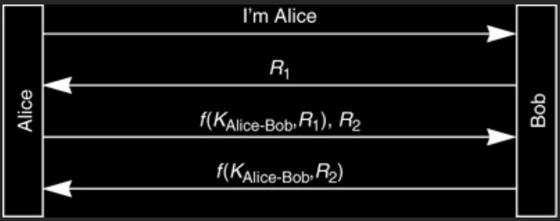

Trudy
Bob
I'm Alice, R_2
R_1, $f(K_{Alice-Bob}, R_2)$
Trudy
Bob
I'm Alice, R_1
R_3, $f(K_{Alice-Bob}, R_1)$

It aims to construct an extra session of authentication, that is useful just for the attacker. The second session will be left incompleted, but is used by the attacker to complete the first one. The schema is the following: Trudy is pretending to be Alice (messages are spoofed), so she sends to Bob "I'm Alice, this is my challenge R_2", Bob is correcting responding with encrypted R_2 and sends to Trudy his nonce R_1. Trudy is not able to correctly respond to Bob's challenge R_1 because she doesn't know the secret key shared between Alice and Bob. So Trudy is starting a new session of authentication saying again "I'm Alice, this is my challenge R_1", but look, Trudy is choosing number R_1 that is just the same challenge that was chosen by Bob in the first session and that Trudy was not able to encrypt, so in this way when Trudy challenges Bob with R_1 Bob will respond with the encrypted R_1 and so now Trudy knows the correct response to the challenge R_1. So now Trudy is able to complete the 1st session. The 2nd session will be not completed. This can raise an alarm, Bob can suspect something.

How reflection attack can be prevented? A way could be that the challenge sent by Alice belongs to a set of numbers and the challenge sent by Bob belongs to another set of numbers, for instance the set of odd numbers and the set of even numbers. Another possible measure is changing the key (key of the initiator different from the key of the responder).

Actually in the reality we do not use this schemas of authentication, we use other schemas that are more robust and that rely on these schemas.

Less optimized mutual authentication:



It is another schema, the number of messages that are exchanged is 4. Less optimized because it uses one more message than the optimized version. A nice exercise would be to check wheter using this schema is still possible to set up the reflection attack. Keep in mind that Trudy can set up an offline-password attack, so she plays as Bob, she sends the challenge and Alice sends her back the correct response, so Trudy by knowing the pair (challenge,response) can guess the secret key with brute force.

Third Party authentication

We can somewhat review this challenge/responde pattern by introducing a third party, that is trusted and that realizes the authentication. It is a very tipical scenario used by companies and enterprises. Tipically this third party is called *Key Deistribution Center* **KDC**, or *Authentication Server*. For brevity we will call this party **C**. This will prevent a quadratic number of keys. Every participant in the network is sharing a secret key only with this server. So with **n** parties there are **n** secret keys that the server maintains. This means that every user, every party has to be registered to the server. Now having this server, we want to implement single or mutual authentication as before. An optional would be to use a session key for short time communications. If the attacker is an insider, he too is sharing with the server a private key, he has some power, but he cannot guess random numbers generated by the parties and cannot be able to get the private keys of other parties and cannot decrypt messages in short time without knowing the key because the attacker is expected not to have high computational power.

We are going to view different schemas of authentication with a third party, slowly converging to the best one.

Schema 1

Presented just for educational purposes, it is very simple and weak. Alice wants to talk with Bob so she sends a message to C saying "I'm Alice, I want to talk with Bob", C choses the session key K and sends Alice two messages that are K encrypted with K_{AC} and K encrypted with K_{BC}. The content of the message is the same, so C is sending to Alice twice the session key K. Then Alice sends to Bob a message constituted by three elements, the message is somewhat like “I’m Alice, I obtained this thing by C, this thing is K_{BC}(K)”. So Bob can decrypt K_{BC}(K) because it has been encrypted by means of his private key, then Bob sends to Alice a message encrypted with the session key, so Alice will be sure that it is really Bob that is using the session key. This schema can be attacked. Assume Trudy can sniff and make *Man in the Middle Attack*, that is Trudy is between Alice and C and between Alice and Bob. So Trudy can intercept messages, and create a session key between her and Alice and another between her and Bob, so Alice and Bob think to talk each other, but actually there is Trudy in the middle that drives the whole conversasion.

Schema 1 (revisited)

There is a modified version of the previous schema, where Alice says to C that she wants to talk with Bob but she says this in an encrypted way (using her private key K_{AC}), so only the Server is able to know with whom Alice wants to talk with, so Trudy cannot guess who is. Also this protocol can be attacked, it is required that Alice and Trudy have communicated at least once, so Trudy can perform a *replay attack* combined with the previous attack.

Needham-Schroeder Protocol

It is a very important protocol that poses the basis for ***Kerberos***. It is used to create a session key between two parties by means of a third party (the Server).

Schema: Alice wants to talk with Bob, so she is choosing a nonce N and she sends to C “I’m Alice, I want to talk to Bob, this is my nonce N”. C generates the session key **K**, and sends to Alice an encrypted message by means of the secret key of Alice, the message contains the nonce, the session key, the identity of Bob, and the encryption of K and A by means of the key of Bob. It is included also A because in this way Bob will know that Alice wants to talk with him. Alice decrypts this message, checks the nonce and B, then sends to Bob the encrypted K and A. Bob decodes, sees that Alice is the other party, then he chooses a nonce N’, and sends to Alice “I am Bob, this is my nonce” encrypted by means of K. Alice can reply with a message encrypted by means of K, and the message says “I’m Alice, this is your nonce decremented by one”.
Notice that Bob is not communicating with C.

Denning & Sacco attack against Needham-Schroeder

Also this protocol can be attacked, the attack is shown on slide 34. The attack is performed using an old session key **K’**. Note that if K’ is compromised, it means that it is an old key, so Trudy can send to Bob K_{BC}(K’,A), it is just a replay attack, Trudy does not need to know K_{BC}! So in order to sum up, the first two messages of the protocol are the original ones, than when Alice sends the encrypted session key to Bob, Trudy starts to play performing a replay attack for which she is able to force Alice and Bob to use an old session key which is compromised and so Trudy, once the corrupted key is established, will be able to decrypt every message and so she will be able to easly perform a man in the middle attack.

There are many possible variants of this attack, and all these variants are important just to let us understanding why we need to introduce more ingredients like timestamps, sequence numbers and so on.
Therefore a *variant* of the protocol has been invented.

Needham-Schroeder Protocol variant

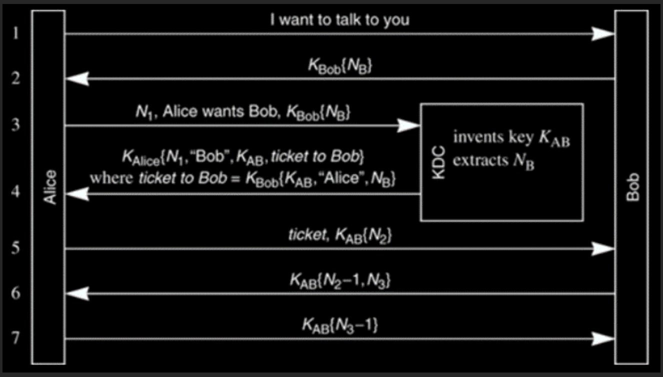
Since the original protocol was weak and we have seen some attacks against it, the protocol has been revisited and here is the new version:

1. A chooses N (nonce) and sends to Bob <A,N>
2. B chooses N’(nonce) and sends to the Server <B,N’, K_{BC}(N,A,t)>
3. The Server sends to A <K_{AC}(B,N,K,t), K_{BC}(A,K,t), N’>
4. A sends to B <K_{BC}(A, K, t), K(N’)>

So in few words, Alice wants to talk to Bob, so she sends him a nonce N. Bob sees this message, invents a nonce N’ and a timestamp t and sends to the server the nonce N’ and the timestamp encrypted by means of

K_{BC}, in this way only B and the server know the timestamp, and the Server will have the power to say if the timestamp is valid or not. Then the server sends to Alice the encrypted session key and the encrypted timestamp by means of K_{AC}, and then another encrypted messante by means of K_{BC} that Alice has to forward to Bob, togheter with the nonce of Bob encrypted with the new session key.

Expanded Needham-Schroeder:



Alice want to talk with Bob, Bob generate a *nonce* and sends it to Alice encrypted by the secret key of Bob, now Alice sends a message to the Server (see content on slide 39 step 3). Then the Server generates the session key $\mathbf{K_{AB}}$ and extracts the *nonce* created by Bob. Then the Server replies to Alice with a message (step 4) that contains the *ticket to Bob*. Alice decrypts in order to get the ticket and sends it to Bob toghether with a new *nonce* encrypted by means of the session key. Bob replies with a message encrypted by the session key. The last message is not really necessary. This last version has inspired **KERBEROS**.