"Let me just spend few words for reminding where we are now. We talked about symmetric encryption, then we talked about data integrity and mac codes, hmac and hashing. After that we considered public key encryption and its applications. Then we saw some usage of public keys for confidentiality but expecially for key exchanging. In addition we saw digital signing. We then concluded the 'public-key infrastructure' topic facing one of the most important issues: i want to remind you this important issue, that is the association between public key and identity. This is the weakest point. The problem of trust! For finalizing this concept and analizing all remaining details we have to study digital certificate (we will study it soon)."

RANDOM NUMBERS

We have already seen that in many practical cases *random numbers* are required for guarantee security of several algorithms. Think of the case you have the best encryption software, using the best symmetric encryption algorithm (AES 256 bit), the most robust one. You need a key, what if the adversary can predict the key? You understand that the robustness of the algorithm becames useless.

Think of the DH key exchange. Alice is choosing a random number (her private key) and Bob is doing the same. In many practical cases both partners are using the same tool for generating random numbers. Even if this is not true and they are using different tools, if the adversary can see the communication and one of the 2 generator is week and the adversary can predict the next number, he can break the next session of communication. All these problems are there because of the bad generation of random numbers.

Random Number Generator

We have two kinds of Random Number Generators:

- RNG = Random Number Generator
- PRNG = Pseudo RNG

What is the difference? The first one, RNG, relies on natural factors and physical phenomenon that can be often considered true random factors, while the second one, PRNG, is just an algorithm that tries to create a *long sequence of apparently random numbers* starting from a certain *key* or *seed,* so all the sequence is completely determined by the *seed*.

There are practical problems in getting random numbers from the nature, also when you get random information you are introducing some biasing in the information (because human use some tools for measuring quantities) and this affects randomness.

Deterministic algorithms: Given an algorithm, if i give it an input i get an output, if i give it the same input it gives me the same output. This means beeing deterministic. Since the algorithm is deterministic, we need some true random input for initializing the state of the algorithm in order to get an unpredictable output.

RNG vs PRNG

Because of all these pactical problems we will be using Pseudo Number Generator algorithms. Real random generators are not practical, or they are expensive. PRNG use an initial value (a seed, that is our key) that has to be kept secret. The generation of the random number will depend on our seed. Therefore giving the seed to the adversary means giving him all the information he needs, because since we don't like security by obscurity our generator is not secret, but the seed has to be! Our security relies on the secrecy of the seed.

Choose of the initial seed: again you need some random input stuff like system data (time, info from the disk, fragmentation, free space, network informations and so on). All this stuff seem to be very random. A good practice could be using a function that mixes several stuff like this. In many cases the current time is not useful, because it is made by small number of bits, then if the adversary knows we are using this method, if he knows for example the day we encrypted it can easly try all the combinations of timestamps of that day, because they are not so many. If you google for attacks to random number generators you find very interesting stuff. Pepole have been using an integer number as initial seed, but in this way there are at most 2^{32} = 65536 possible seeds. If the granularity of the time is in the order of 10 milliseconds, if the attacker knows the hour we encrypted the message the number of seeds he has to try is:

60x60x100 = 360000

Slide 8: the *Nescape 1.1 seeding & key generation*. The algorithm is composed by two parts, the first one is just the part that *pseudo-randomly* chooses the seed, the second is a function that is called whenever we need the next number of the sequence of the random numbers. Howerver, 2 vulnerabilities are present:

1. The seed is very small and it is very simple to predict the initial seed
2. The algorithm uses MD5 that is a broken hashing function

The output of MD5 is 128-bit, so today it is weak (with the birthday attack we run 2^{64} iterations more or less).

"Why we are talking about MD5? Because still today is very popular. Do you know what is digital forensics? It is a process that is used by experts when they have to analize the state of a computer or the content of a disk. Still today there are pseudo experts that extract information from disks and in order to proove data integrity they offer MD5."

Random Number bug in Debian (2006)

You know openSSL, it is a very important library in order to provide cryptographical functions. Debian had a package using openSSL, then a debian tool called *purify* gave a warning on a line of code of that package, stating that some variable may had not been intialized. So the person that developed the package tought to remove them in order to get rid of the warning. What happenes was that the only used "random" value for the seed was the PID of the process. On Linux platforms, a PID is in the range [0,32768], and therefore that bug caues a strong weakness of the system.

BSI Evaluetion

Criteria for evaluating the quality of a PNRG. There are 4 criteria:

1. K1: a sequence of random numbers with a low probability of containing identical consecutive elements
2. K2: actually it is a list of requirements on statistical tests (see slide 19)
3. K3: it should be impossible for the attacker to guess, for any given sub-sequence, any previous or future values in the sequence, nor any inner state of the generator
4. K4: it should be impossible for the attacket to guess from an inner state of the generator, any prevous numbers in the sequence or any previous inner generator states

Cryptographically Secure Pseudo Random Number Generators (CSPRNG)

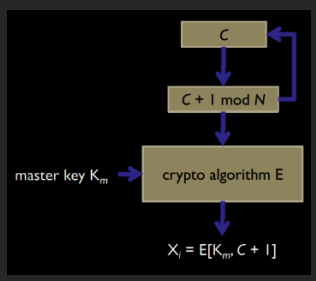
2 qualities are required:

- Next-bit test
- State compromise extentions

Statistical definition: there is no polynomial algorithm that is able to distinguish with probability > 0.5 a truly random sequence from a pseudo-random sequence.

Examples of PRNG

Cipher of a Counter



This generator has been used for long time. There is just a counter and the counter is incrementing mod N. This is offering the possibility to have a limited number of pseudo rundom numbers, determined by N. The approach is : encrypt some integer number by means of an encryptor, the encryptor will be making of course usage of a key k_m, then the output is the pseudo random number. It is clear that even in the case some generated number is compromised, the next number will be unknown due to the usage of the key.

In the last block, do we need a mod operation? It depends on the input that goes into the last block, because if the input that goes into the last block is bigger than the maximum input that is supported by the last block we need a mod operation inside it. This is why people use small number of N (tipically 64 bits). Are 64 bit enough? In many cases yes, sometimes not and you have to reorganize the structure.

RSA based generator

It is a generator stronger than the previous one and as the name suggests it is based on RSA. The generator states:

Take two primes *p* and *q*, than compute *n=p*q*, then find a number *e* such that *e* is coprime with Φ(n) = (p-1)*(q-1). Then a seed *z* is chosen, and then starts a loop where the current random number is previous one raised to the power of *e* and then mod n. Obviously we start from the seed *z*.

Last slide: that generator is not good because if the adversary discovers one generated number, he will know all next generated numbers.