Lecture 10: Key-Distribution and Random-Number Generation Issues Related to Symmetric Key Cryptography

Lecture Notes on "Introduction to Computer Security"

by Avi Kak (kak@purdue.edu)

February 22, 2007

Goals:

- Why might we need **key distribution centers**?
- **Master keys** vs. **Session keys**
- Hierarchical and decentralized key distributions
- Review of approaches for generating pseudorandom numbers needed for system-generated encryption keys, etc.

The Need for Key Distribution Centers

- Let's say we have a large number of people, processes, or systems that want to communicate with one another in a secure fashion. Let's further add that this group of people/processes/systems is not static, meaning that the individual entities may join or leave the group at any time.

- A simple-minded solution to this problem would consist of each party physically exchanging an encryption key with every one of the other parties. Subsequently, any two parties would be able to establish a secure communication link using the encryption key they possess for each other. **This approach is obviously not feasible for large groups of people/processes/systems, especially when group membership is ever changing.**

- A more efficient alternative consists of providing every group member with a single key for securely communicate with a **key distribution center** (KDC). This key would be called a **master key**. When A wants to establish a secure communication link with B, A requests a **session key** from KDC for communicating with B.

- In implementation, this approach must address the following issues:

– Assuming that A is the initiator of a session-key request to KDC, when A receives a response from KDC, how can A be sure that the sending party for the response is indeed the KDC?

– Assuming that A is the initiator of a communication link with B, how does B know that some other party is not masquerading as A?

– How does A know that the response received from B is indeed from B and not from someone else masquerading as B?

– What should be the lifetime of the session key acquired by A for communicating with B?

A Protocol for Acquiring and Using a Session Key

Assumptions: A party named A wants to establish a secure communication link with another party B. Both the parties A and B possess **master keys** K_A and K_B, respectively, for communicating privately with a **key distribution center** (KDC). Now A engages in the following protocol:

- Using K_A, A sends a request to KDC for a **session key** intended specifically for communicating with B.

- The message sent by A to KDC includes A's network address (ID_A), B's network address (ID_B), and **a unique session identifier**. The session identifier is a **nonce** — short for a "number used once" — and we will denoted it N_1. The primary requirement on a nonce — a random number — is that it be unique to each request sent by A to KDC. The message sent by A to KDC can be expressed in shorthand by

$$E(K_A, \quad [ID_A,\ ID_B\ , N1])$$

  where $E(.,.)$ stands for encryption of the second-argument data block with a key that is in the first argument.

- KDC responds to A with a message encrypted using the key K_A. The various components of this message are

  - The session-key K_S that A can use for communicating with B.

  - The original message received from A, including the nonce used by A. This is to allow A to match the response received from KDC with the request sent. Note that A may be trying to establish multiple simultaneous sessions with B.

  - A "packet" of information meant for A to be sent to B. This packet of information, encrypted using B's master key K_B includes, again, the session key K_S, and A's identifier ID_A. (Note that A cannot look inside this packet because A does **not** have access to B's master key K_B.

- The message that KDC sends back to A can be expressed as

$$E(K_A, \ \ [K_S, \ ID_A, \ ID_B \ , N1, \ E(K_B, \ [K_S, \ ID_A] \ ])$$

- Using the master key K_A, A decrypts the message received from KDC. Because only A and KDC have access to the master key K_A, A is certain that the message received is indeed from KDC.

- A keeps the session key K_S and sends the packet intended for B to B. This message is sent to B unencrypted by A. But note that it was previously encrypted by KDC using B's master key K_B. **Therefore, this first contact from A to B is protected from eavesdropping.**

- B decrypts the message received from A using the master key K_B. B compares the ID_A in the decrypted message with the sender identifier associated with the message received from A. By matching the two, B makes certain that no one is masquerading as A.

- B now has the session key for communicating securely with A.

- Using the session key K_S, B sends back to A a nonce N_2. A responds back with $N_2 + 1$, using, of course, the same session key K_S. This way B knows that it did not receive a first contact from A that A is no longer interested in. This is also a protection against a "replay" attack.

- A replay attack is a form of **network attack** in which a third party — C — eavesdrops on the communications between A and B. Let's say that C intercepts the first-contact message that B received from A. Now the question is: Would C be able to pose as B during a subsequent attempt by A to initiate a session with B? Let's assume that C has somehow gotten hold of B's master

key K_B. Nonces, being one-time tokens, make such masquerades more difficult. (After we have reviewed how message authentication codes are used, we will review this issue in greater detail.)

- The message sent by B back to A can be expressed as

$$E(K_S,\ N2)$$

And A's response back to B as

$$E(K_S,\ N2\ +\ 1)$$

- This exchange of message is shown graphically in the figure on the next page. **A most important element of this exchange is that what the KDC sends back to A for B can only be understood by B.**

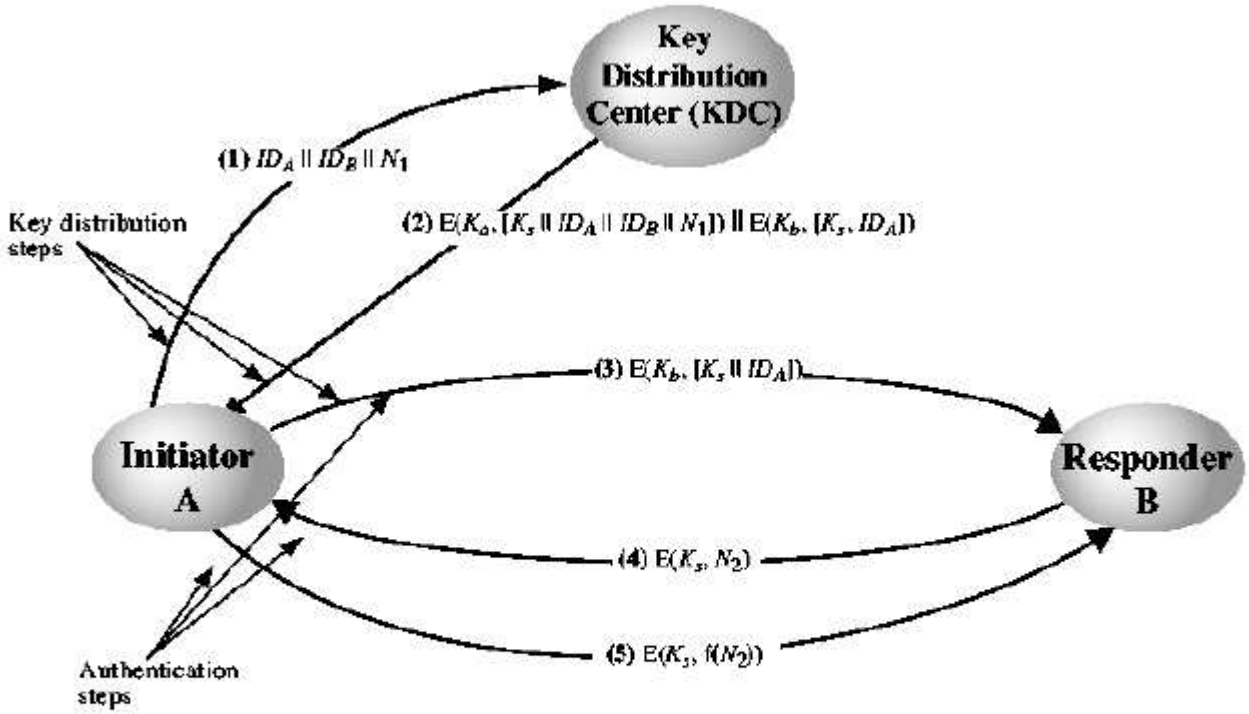

Figure 7.9 Key Distribution Scenario

This figure is from Chapter 7 of Stallings: "Cryptography and Network Security", Fourth Edition

Some Variations on the KDC Approach to Key Distribution

- Establishing a hierarchy of key distribution centers:
  - It is not practical to have a single KDC service very large networks or network of networks.
  - One can think of KDC's organized hierarchically, with each local network serviced by its own KDC, and a group of networks serviced by a more global KDC, and so on.
  - A local KDC would distribute the session keys for secure communications between users/processes/systems in the local network. But when a user/process/system desires a secure communication link with another user/process/system in another network, the local KDC would communicate with a higher level KDC and request a session key for the desired communication link.
  - Such a hierarchy of KDCs simplifies the distribution of master keys.

- A KDC hierarchy also limits the damage caused by a faulty or subverted KDC.

- Decentralization of key distribution:

  - It is possible to completely decentralize key distribution by storing at every node of a network the master keys needed for each of the other N nodes in a network. Therefore, each node will store $N - 1$ master keys.

  - This is still more secure than using those $N - 1$ master keys directly as session keys. Since a master key will only be used for establishing a session key, the messages encrypted using the master keys will be short, making difficult their cryptanalysis. The session keys, on the other hand, will change frequently.

Random Number Generation

Using cryptography to make networks more secure requires that we be able to generate random numbers on the fly:

- **The session keys that a KDC must generate on the fly are nothing but a sequence of random numbers.** A session key may, for example, be a sequence of integers with each integer being in the 0–255 range, both ends inclusive. Each integer in such a sequence would be a randomly generated byte. For the purpose of transmission over character-oriented channels, each byte in such a sequence could be represented by its two hex digits. Therefore, a 128 bit session key would be a string of 32 hex digits.

- The nonces that are exchanged during handshaking between a host and a KDC and amongst hosts are also random numbers.

- As we will see later, random numbers are also needed for the RSA public-key encryption algorithm.

When are Random Numbers Truly Random

- To be considered truly random, a sequence of numbers must exhibit the following two properties:

  Uniform Distribution: This means that all the numbers in a designated range must occur equally often.

  Independence: This means that if we know some or all the number up to a certain point in a random sequence, we should not be able to predict the next one (or any of the future ones).

- Truly random numbers are a mathematical abstraction. Truly random numbers can only be generated by physical phenomena (such microscopic phenomena as thermal noise, and such macroscopic phenomena as cards, dice, and roulette wheel).

- Algorithmically generated random numbers can only be approximations to true random numbers with regard to the two properties mentioned above.

- Algorithmically generated random numbers are called **pseudo-random numbers**.

Pseudorandom Number Generators (PRNGs): Linear Congruential Generators

- This has been a common approach for generating pseudorandom numbers.

- Starting from a seed X_0, a sequence of (presumably pseudorandom) numbers X_0, X_1, ..., X_i, ... is generated using the recursion:

$$X_{n+1} \quad = \quad (a \cdot X_n \quad + \quad c) \ \text{mod} \ m$$

where

m	*the modulus*	$m > 0$
a	*the multiplier*	$0 < a < m$
c	*the increment*	$0 \leq c < m$
X_0	*the seed*	$0 \leq X_0 < m$

- The values for the numbers generated will be in the range $0 \leq X_n < m$.

- As to how random the produced sequence of numbers is depends critically on the values chosen for m, a, and c. For example, choosing $a = c = 1$ results in a very predictable sequence.

- **Should a previously generated number be produced again, what comes after the number will be a repeat of what was seen before.** That is because for a given choice of m, c, a, the next number depends only on the current number. Consider the case when $a = 7$, $c = 0$, $m = 32$. The sequence of number produces is $\{7,\ 17,\ 23,\ 1,\ 17,\ 23, ...\}$. The period in this case is only 4.

- Since the "randomness" property of the generated sequence of numbers depends so critically on m, a, c, people have come up with criteria on how to select values for these parameters:

  - To the maximum extent possible, the selected parameters should yield a **full-period sequence** of numbers. The period of a full-period sequence is equal to the size of the modulus. Obviously, in a full-period sequence, each number between 0 and $m - 1$ will appear only once in a sequence of m numbers.

    It has been shown that when m is a prime and c is zero, then for certain value of a, the recursion formula shown above is guaranteed to produce a sequence of period $m - 1$. Such a sequence will have the number 0 missing. But every number n, $0 < n < m$, will make exactly one appearance in such a sequence.

  - The sequence produced must pass a suite of statistical tests

meant to evaluate its randomness. These tests measure how uniform the distribution of the sequence of numbers is and how statistically independent the numbers are.

- Commonly, the modulus m – we want it to be a prime – is chosen so that it is also the largest positive integer value for a system. So for a 4-byte signed integer representation, m would commonly be set to $2^{31} - 1$. With $c = 0$, our recursion for generating a pseudorandom sequence then becomes

$$X_{n+1} = (a \cdot X_n) \mod (2^{31} - 1)$$

- Earlier we said that when m is a prime and c is zero, then certain values of a will guarantee an output sequence with a period of $m - 1$. A commonly used value for a is $7^5 = 16807$.

- Statistical properties of the pseudorandom numbers generated by using $m = 2^{31}-1$ and $a = 7^5$ have been analyzed extensively. It is believed that such sequences are statistically indistinguishable from true random sequences consisting of positive integers greater than 0 and less than m.

- **But are such sequences cryptographically secure?**

- A pseudorandom sequence of numbers is **cryptographically secure** if an attacker cannot predict the next number from the numbers already in his/her possession.

- When **linear congruential generators** are used for producing random numbers, **the attacker only needs three pieces of information to predict the next number from the current number**: m, a, c. The attacker may be able to infer the values for these parameters by solving the simultaneous equations:

$$\begin{aligned} X_1 &= (a \cdot X_0 + c) \bmod m \\ X_2 &= (a \cdot X_1 + c) \bmod m \\ X_3 &= (a \cdot X_2 + c) \bmod m \end{aligned}$$

  Just as an exercise assume that $m = 16$, $c = 0$, and $a = 3$. Assuming X_0 to be 3, set up the above three equations for the next three values of the sequence. These values are 9, 11, and 1. You will see that it is not that difficult to infer the value for the parameters of the recursion.

- The upshot is that even when a PRNG produces a "good" random sequence, it may not be secure enough for cryptographic applications.

- A pseudorandom sequence produced by a PRNG can be made more secure from a cryptographic standpoint by restarting the sequence with a different seed after every N numbers. One way to do this would be to take the current clock time modulo m as a new seed after every so many numbers of the sequence have been produced.

Cryptographically Secure PRNGs: The ANSI X9.17 Algorithm

- This technique for generating pseudorandom numbers is used in many secure systems for financial transactions. **This technique is also used in PGP.**

- As shown on slide 21, this PRNG is driven by two encryption keys and two special inputs that change for each output number in a sequence.

- Each of the three "EDE" boxes shown on slide 21 stands for the two-key 3DES algorithm. As you will recall from Lecture 9, the two-key 3DES algorithm carries out a DES encryption, followed by a DES decryption, and followed by a DES encryption. The acronym EDE means "encrypt-decrypt-encrypt".

- The two inputs are: (1) A 64-bit representation of the current date and time (DT_j); and (2) A 64-bit number generated when the previous random number was output (V_j). The PRNG is initialized with a seed value for V_0 for the very first random number that is output.

- All three EDE boxes shown in the figure use the same two 56-bit

encryption keys K_1 and K_2. These two encryption keys stay the same for the entire pseudorandom sequence.

- The output of the PRNG consists of the sequence of pairs $(R_j,\ V_{j+1})$, $j\ =\ 0,\ 1,\ 2, ...$, where R_j is the j^{th} random number produced by the algorithm and V_{j+1} the input for the $(j+1)^{th}$ iteration of the algorithm. From the figure on the next slide, the output pair $(R_j,\ V_{j+1})$ is given by

$$R_j = EDE\left([K_1, K_2]\,,\ \ [V_j \otimes EDE([K_1, K_2]\,,\ DT_j)]\right) \quad (1)$$
$$V_{j+1} = EDE\left([K_1, K_2]\,,\ \ [R_j \otimes EDE([K_1, K_2]\,,\ DT_j)]\right) \quad (2)$$

  where $EDE([K_1, K_2],\ X)$ refers to the encrypt-decrypt-encrypt sequence of 3DES using the two keys K_1 and K_2.

- The following reasons contribute to the cryptographic security of this approach to PRNG:

  – We can think of V_{j+1} as a **new seed** for the next random number to be generated. This seed cannot be predicted from the current random number R_j.

  – Besides the difficult-to-predict pseudorandom seed for each random number, the scheme uses one more independently

specified pseudorandom input — an encryption of the current date and time.

- Each random number is related to the previous random number through multiple stages of DES encryption. An examination of Equation (1) on the previous slide shows there are **more than two EDE encryptions** between two consecutive random numbers. If you could say from Equation (2) that there exists one EDE encryption between a random number and the seed for the next random number, then it would fair to say that there exist **three EDE encryptions** between two consecutive random numbers. Since one EDE encryption amounts to three DES encryptions, we can say that there exist **nine DES encryptions** between two consecutive random numbers, making it virtually impossible to predict the next random number from the current random number.

- Even if the attacker were to somehow get hold of the current V_j, it would still be practically impossible to predict V_{j+1} because there stand at least two EDE encryptions between the two.

- Is there a price to pay for the cryptographic security of ANXI X9.17? Yes, it is much slower way to generate pseudorandom numbers. That makes this approach unsuitable for use in Monte Carlo simulations and other situations that require randomized inputs.

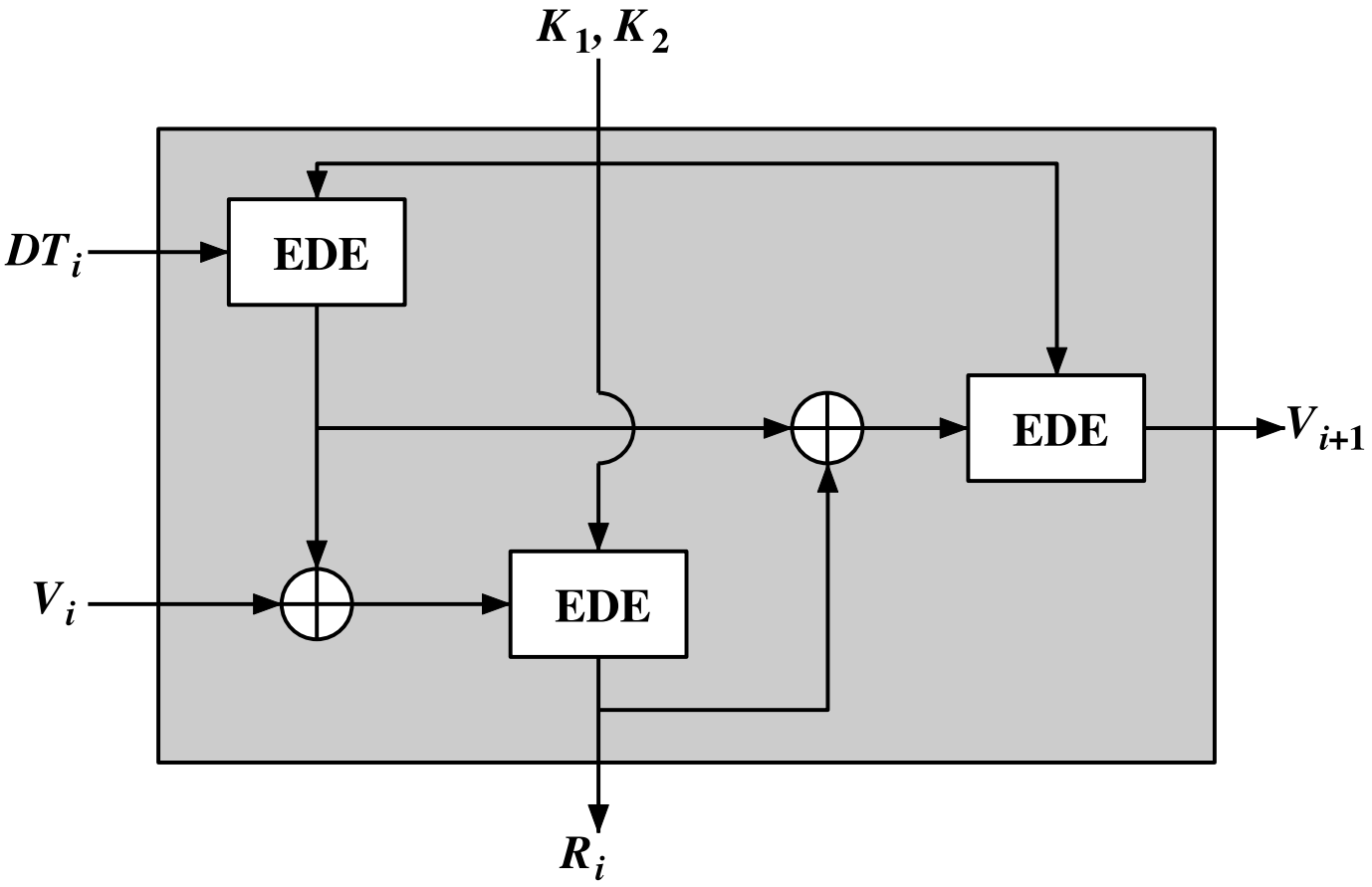

Figure 7.14 ANSI X9.17 Pseudorandom Number Generator

This figure is from Chapter 7 of Stallings: "Cryptography and Network Security", Fourth Edition

Cryptographically Secure PRNGs: The Blum Blum Shub Generator (BBS)

- This is another cryptographically secure PRNG. This has probably the strongest theoretically proven cryptographic security.

- The BBS algorithm consists of first choosing two large prime numbers p and q that both yield a remainder of 3 when divided by 4. That is

$$p \equiv q \equiv 3 \pmod 4$$

  For example, the prime numbers 7 and 11 satisfy this requirement.

- Let

$$n = p \cdot q$$

- Now choose a random number s that is relatively prime to n. (This implies that p and q are **not** factors of s.)

- The BBS generator produces a pseudorandom sequence of bits B_j according to

$$
\begin{aligned}
X_0 &= s^2 \mod n \\
\\
for\ i &= 1\ to\ \text{inf} \\
X_i &= (X_{i-1})^2 \mod n \\
B_i &= X_i \mod 2
\end{aligned}
$$

- Note that B_i is the least significant bit of X_i at each iteration.

- Because BBS generates a pseudorandom bit stream directly, it is also referred to as a **cryptographically secure pseudorandom bit generator** (CSPRBG).

- By definition, a CSPRBG must pass the **next-bit test**, that is there must not exist a polynomial-time algorithm that can predict the k^{th} bit given the first $k-1$ bits with a probability significantly greater than 0.5. BBS passes this test.