EECS-122 lecture notes on Network Security

Stavros Tripakis

1 Introduction

Network security was not probably one of the top priorities of the designers in the first years of the Internet. Recently, however, the increase in number of users and critical applications such as e-commerce have created a demand for Internet security. This has given rise initially to the use of firewalls (routers that isolate a network from the rest of the Internet and filter the incoming and outgoing traffic according to some rules fixed by the administrator), and more recently to more sophisticated security protocols such as Kerberos or SSL (secure socket layer).

What do we mean by network security ? Simply stated, we want our network security mechanism to make sure that whenever two parties (human users, or applications) A and B engage in a transaction through the network, the following holds:

If B receives message M from A, then:

1. A (not somebody else) indeed sent M (not something else) to B (not to someone else); and
2. Nobody else knows that A sent M to B.

Three requirements emerge out of the above statement:

- The network security mechanism must provide means of *authentication* (or *identification*) of the parties involved in a transaction. In other words, one party must be able to check whether the other party is really who it claims to be. Passwords are used for authentication in computer accounts: to prove that you are who you claim to be in your login-id, you must provide the password that goes with this login-id (it is assumed that only you know this password). Passwords cannot be used directly for network authentication, since an eavesdropper can easily read a password traveling through the network.
- The network security mechanism needs to ensure the *integrity* of the message M. That is, it should make sure that an intruder does not alter the message, or if it does, then the recipient B can tell that the message has been altered (and perhaps discard it).
- The network security mechanism needs to ensure the *privacy* of the message, of the communicating parties, or of both. This means that a third party should not be able to find out the content of M, or the identity of A or B, or even stronger, should not learn that a communication between A and B took place.

What do we mean by attacks ? We have talked about "eavesdroppers", "intruders" and "third parties", without making precise who they are and what they do. Attacks can be formulated mathematically, so that precise statements can be proven about security systems. On the other hand, in order to formulate something mathematically, you have to imagine already what this "something" is. It turns out that imagining all possible attacks is a very hard thing to do, but this is precisely why cryptography and security in general are very interesting subjects !

In these notes, let us be informal and say that an attack includes the following:

- eavesdropping: a third party (not A or B) reads a message being sent from A to B;
- blocking: a third party removes from the network a message destined for B;
- sending: a third party C sends some message, that it has removed and possibly changed (this includes so-called *replay* of messages) or that it has generated from scratch, to A or B;
- masquerading: a third party C can pretend to be someone else (e.g., can change the IP addresses of its packets).

An example of attack that includes all of the above is the so-called "person-in-the-middle" attack, where a third party C interferes in the communication between a client A and a server B. C blocks the requests from A to B and the responses from B to A. C pretends to B to be A (i.e., plays the role of the client for B) and pretends to A to be A (i.e., plays the role of the server for A).

Another common attack is the so-called "replay" attack. Here, a third party C first reads messages exchanged between A and B and (perhaps later, after the transaction between A and B is over) sends some of these messages again to A or B. Notice that this attack is possible without C actually knowing what the exact content of a message it sends is. The message can be encrypted using some of the mechanisms we describe below. Still, C can simply send the message, without having deciphered its content.

We will see more examples of attacks when we talk about security protocols.

Security mechanisms and protocols. The basic security mechanisms are: *cryptography* (used mainly for privacy, but also for authentication and integrity) and *hashing* (used for integrity). Cryptographic mechanisms use *keys* for the encryption and decryption of messages, so, *key generation* and *key distribution* are problems to be solved as well. In the case of public-key cryptography (see below) *certification* of public keys (how to be sure that the key really belongs to some-one claiming it does) is also an issue, discussed below.

A *security protocol* typically involves all or most of the above mechanisms, in order to meet all the requirements.

We first discuss cryptography and hashing, then go on to explain how they are used for privacy, authentication and integrity.

2 Cryptography and Hashing

The idea behind cryptography is the following. When A wants to send a message M (the *plaintext*) to B, instead of transmitting M *in the clear* (i.e., as is), A encodes the message using a function f

and transmits $f(M)$ (the *ciphertext*). At the other end, B decodes $f(M)$ using the inverse of f and retrieves $M = f^{-1}(f(M))$.

Obviously, f should be known to A and f^{-1} should be known to B. We might require these functions to be secret, that is, not known by anyone else except A or B. However, this is impractical. For each pair of parties A and B, there should be a unique, secret pair of functions f and f^{-1}. In practice, f and f^{-1} are implemented by some algorithms, in order to be used in computer systems. Requiring a different function each time means requiring a different algorithm each time, which is very costly (if not impossible) to achieve in practice. Moreover, should the functions ever become compromised (known to others), they should be replaced, which adds to the initial cost.

Therefore, the encryption and decryption algorithms are in practice functions of two arguments, the message (plaintext or ciphertext) and a *key*. These algorithms are divided in two basic categories (presented below), depending on how many keys they use and the assumptions they make about these keys. The intruder may know the algorithm, but it does not know the key. The only way to break the code is to search all possible keys. In fact, it might be necessary to search also all possible plaintexts, given that the intruder knows only the ciphertext and the encryption and decryption algorithms. However, additional information on the plaintext (e.g., it is English text) might make it easy for the intruder to guess it, or be highly confident that it has found the right key, once the decrypted ciphertext with the guessed key gives back a plausible plaintext.

2.1 Secret-key cryptography

In secret-key cryptography, there is a key $K_{A,B}$ for each pair of parties A—B. $K_{A,B}$ should be known only by A and B. To encrypt a message M, A computes $f(M, K_{A,B})$. To decipher an encrypted message E, B computes $f'(E, K_{A,B})$. Obviously, it should hold that: $f'(f(M, K_{A,B}), K_{A,B}) = M$. The functions (algorithms) f and f' themselves might be known. The fact that the key is secret makes it difficult for an eavesdropper, who knows $f(M, K_{A,B})$, to recover M without knowing $K_{A,B}$.

Secret-key cryptography is sometimes also called *symmetric-key* cryptography. Examples of secret-key cryptographic methods are DES and IDEA (see [12]).

2.2 Public-key cryptography

In public-key cryptography, each party B has two keys, a *private* key $PrivK_B$ and a *public* key $PubK_B$. $PrivK_B$ is known only to B, whereas $PubK_B$ is known to anyone. When A wants to send M to B, it computes and sends $f(M, PubK_B)$. To decipher an encrypted message E, B computes $f'(E, PrivK_B)$. Obviously, it should hold that: $f'(f(M, PubK_B), PrivK_B) = M$. In most cases, it is also the case that: $f'(f(M, PrivK_B), PubK_B) = M$. That is, the private and the public key can be used interchangeably.

As before, f and f' themselves are typically known. An eavesdropper who knows $f(M, PubK_B)$ cannot easily recover M without knowing $PrivK_B$. Notice, however, that the space of all possible messages M is assumed to be very large (this is typically the case). Otherwise, an intruder wishing to find M, and having observed $X = f(M, PubK_B)$, would compute $X_i = f(M_i, PubK_B)$, for all possible messages M_i, and check for each i whether $X_i = X$. When $X_j = X$ for some j, the intruder can conclude that $M = M_j$. This is not practical when the space of all possible messages is extremely large.

An example of public-key cryptographic method is RSA (see [12]).

2.3 Comparison of secret-key and public-key cryptography

The main problem in secret-key cryptography is for A and B to agree upon a secret key, without revealing the key to someone else. For instance, one of them, say A, could generate (say, randomly) a key and try to send it securely to B. It looks like a snake biting its tail, since to send the key securely, you need cryptography in the first place ! This is why secret-key cryptography is typically used in combination with public-key cryptography (see below).

Public-key cryptography does not present problems in key distribution. B can generate $PrivK_B$ and $PubK_B$ on its own, and then distribute $PubK_B$ in-the-clear, to anyone it wishes to talk with. A potential problem, however, rises from the fact that someone other than B, say C, can pretend to be B, and give its own public key, $PubK_C$. Then, A would think it's talking to B and encrypt the messages using $PubK_C$. C can decrypt these messages, since it has $PrivK_C$. To solve this problem, certification of public keys is used (see below).

Performance should also be considered when talking about cryptographic techniques. Secret-key methods are generally computationally cheaper than public-key methods. For example, DES uses a 56-bit key and the encryption/decryption of a message is done using boolean operations like register-shift, permutations, XOR, and so on, which can be implemented very efficiently. RSA, on the other hand, uses very long keys (e.g., 512 bits) and requires operations like exponentiation which are expensive. This is why public-key methods are typically used only in the beginning of a transaction, in order for the parties to securely establish a *session key* K. Then, during the transaction, data is encrypted using K and a secret-key method like DES.

Notation. In the sequel, instead of $f(M,K)$ or $f'(E,K')$, we use $K(M)$, $K'(E)$, and so on. In particular, in secret-key cryptography, we write $K(K(M)) = M$. In public-key cryptography, we write:

$$PrivK_B(PubK_B(M)) = PubK_B(PrivK_B(M)) = M.$$

When we present security protocols, we use the notation $A \to B : M$ to mean A sends message M to B (it does not mean that B receives M, there might be an attack).

Using public-key cryptography to establish a session key. Suppose A wants to establish a session key with B, so that A and B can then use this key along with a secret-key method to exchange private messages. Then, the following protocol can be used:

- A : generates random key K.
- $A \to B : PubK_B(K)$.
- B : $PrivK_B(PubK_B(K)) = K$.

At the end, both A and B (and only them) know K.

Why use public-key cryptography just for establishing a secret key, instead of using it for encryption of messages between A and B during the whole duration of their conversation? One reason is that B might not know (nor want to find out) the public key of A: B would need this key if during the conversation, not only A sends messages to B, but also B to A. A more important reason is that public-key cryptography is computationally more costly than secret-key cryptography. During the (long) duration of the discussion between A and B, a large amount of CPU power would be spent encrypting and decrypting the (long) messages that A and B exchange, if a method such as RSA were used. Encrypting and decrypting a short message like K using RSA shouldn't be a problem.

2.4 Breakable and unbreakable cryptographic methods, and why the one-time pad is not used in network security

We may wonder whether the notion of an unbreakable cryptographic method exists mathematically. Such a notion exists indeed, but here we restrict ourselves to an informal description of the results. Intuitively, a method is unbreakable if the ciphertext E reveals no information about the plaintext M. Said differently, knowing E does not "help" at all in finding M: the probability of finding M knowing E is equal to the probability of finding M without knowing E, that is, equal to the probability of finding (the correct) M by a random choice. Of course, the probability of finding M cannot be 0, since an eavesdropper can always make a random choice: the issue is how confident the eavesdropper can be on this choice.

Let us give some examples.

Why is the character-substitution code breakable? Consider a *character-substitution* code, used to encrypt English text. In this code, there is a table T which gives a one-to-one mapping of characters to characters, e.g., $T(a) = \$$, $T(b) = \#$, etc. A message $M = x_1x_2\cdots x_k$ is encrypted as $E = T(x_1)T(x_2)\cdots T(x_k)$, which is decrypted by computing $T^{-1}(T(x_i)) = x_i$. This is not an unbreakable code. Indeed, knowing E (and the fact that this is an English-text message and that a character-substitution code has been used for encryption) helps in recovering T, and then M. We can try to guess T by choosing a corresponding letter for each character that appears in E. After many trials, we can come up with a T such that $T^{-1}(E)$ is indeed a plausible English phrase. Now there is a considerable probability that T is indeed correct. We can verify whether this is so by eavesdropping subsequent messages E', E'' and so on, each time improving our guessing.

The one-time pad. There is a code, however, that can be proven to be unbreakable. This code is called *one-time pad* (it might as well be called one-time key). Suppose that the message M is a bit-word (if it is text, use ASCII to translate it into bits) of length n. Let R be a random bit-word of length n as well: R is random in the sense that there is equal probability $\frac{1}{2}$ that a bit in R is 0 or 1. You might say that R is generated by flipping n coins, and letting the i-th bit be 1 if the i-th coin comes heads, 0 if it comes tails. Then, M is encrypted as $E = M \oplus R$, where $\oplus$ denotes the logical bit-wise XOR.

It can be shown that if R is random, then so is E, that is, the probability of a bit of E being 0 or 1 is the same, $\frac{1}{2}$. Therefore, knowing E does not help us at all in recovering M. We just as might have generated E ourselves by flipping coins. Notice that the same "letter" (i.e., pattern of bits) that appears in two or more different positions in M, will be encoded as different patterns in E, since the corresponding patterns in R are not the same (they are random). Therefore, even knowing that M was generated by ASCII text, that is, by blocks of 7 bits each representing a letter, does not help in finding which bit-pattern in E corresponds to which letter. E might have been produced by M and R, or equally well by another M' and R'.

The one-time pad is no longer unbreakable if the same pad (key) R is used more than once. Knowing $E_1 = M_1 \oplus R$ and $E_2 = M_2 \oplus R$, we can first of all compute $E_1 \oplus E_2 = (M_1 \oplus R) \oplus (M_2 \oplus R) = (M_1 \oplus M_2) \oplus (R \oplus R) = (M_1 \oplus M_2) \oplus 0 = M_1 \oplus M_2$. Thus, we now know $M_1 \oplus M_2$, new information which we didn't have before. Now we can start guessing R, each time retrieving M_1 and M_2 from E_1 and E_2 and (assuming well-known ASCII encoding) verifying whether M_1 and M_2 are sensible English phrases.

Why are DES or RSA theoretically breakable? We said that if R is used once, then the one-time pad is unbreakable, that is, knowing $M \oplus R$ does not help in recovering M. Why doesn't the same apply to, say, DES? That is, where does knowing $K(M)$ help in finding M? Knowing $E = K(M)$ indeed helps to find M. An intruder does not know K, but can start trying all possible keys $K_i, i = 1, 2, \ldots$, computing each time $M_i = K_i(E)$. If the intruder finds some K_i such that M_i is a plausible message, then the intruder can be very confident that K_i is indeed the right key and M_i is indeed M. This is because the space of all possible keys is much smaller than the space of all possible messages, therefore, the probability of having two keys $K_i \neq K_j$ such that $M_i = K_i(E)$ and $M_j = K_j(E)$ are both plausible messages is very small.

Why are theoretically breakable codes used? Notice that we are talking about theoretically breakable codes. The fact that it might take a very large number of tries to break them is irrelevant to the theoretic result.

In any case, why isn't the one-time pad used for network security? First of all, were the two parties to use each pad (key) R only once, they would have to agree upon a new R for each message exchange, which is impractical: if there is a secure way to transmit R, then why not transmit the message instead, which is just as long ! If R was to be used multiple times, the one-time pad is no longer theoretically unbreakable. There is no reason to use it, since other methods exist which are theoretically breakable, but very hard to break in practice.

Other disadvantages of the one-time pad include the fact that R has to be truly random (not pseudo-random, in which case it could be guessed). Also, if for some reason the intruder manages to find a plaintext M, then the key R can be recovered immediately from the ciphertext, and used to decrypt subsequent messages. In good cryptographic methods (such as RSA or DES) it is hard to find the key K, even if both a plaintext M and the corresponding ciphertext $K(M)$ are known.

3 Public key certification

As mentioned above, A cannot believe anyone who claims: "I am B, my public key is K". It might be C, pretending to be B, and using $K = PubK_C$, which would allow C to decrypt the messages sent from A to B.

To get the public key of B, A contacts a *trusted third party*, called *certification authority* (CA). CA sends to A a message of the form: "the public key of B is $PubK_B$". This message cannot be sent in the clear, otherwise C could remove it and change it to read: "the public key of B is $PubK_C$". Therefore, the message is sent encrypted by the private key of the CA. That is, the CA sends: $PrivK_{CA}(B, PubK_C)$. It is assumed that the CA does not lie (that's why it is trusted). It is also assumed that A knows the (real) public key of the CA (perhaps installed by the administrator in a secure database), so A can decrypt the above message and recover the public key of B. Finally, an intruder C cannot pretend to be B, since it cannot generate $PrivK_{CA}(B, PubK_C)$ (C would have to know $PrivK_{CA}$).

A single CA is not scalable: each time a user wanted to find out the public key of another (perhaps remote) user, it would have to contact the CA. This is why (in the X.509 standard) a hierarchy of CAs is proposed. For example, consider a root CA and two subordinate CAs, CA_1 and CA_2. Assume A belongs to CA_1 and B to CA_2. To get the public key of B, A first contacts CA_1, which replies by saying something like "please contact the root CA to get this info, here's the root CA's public key": $PrivK_{CA_1}(CA, PubK_{CA})$. A goes on to contact CA who replies by forwarding A to CA_2: $PrivK_{CA}(CA_2, PubK_{CA_2})$. Finally, A contacts CA_2 who hands back the public key of B: $PrivK_{CA_2}(B, PubK_B)$.

Notice that there is a "chain" of certificates (and correspondingly, of trust): CA_1 certifies the public key of CA; CA certifies the public key of CA_2; CA_2 certifies the public key of B. A trusts CA_1, therefore also CA, therefore also CA_2.

4 Authentication

Authentication is an important part of any security system. It allows a party B to be confident that another party claiming to be A is indeed A. Here, A might mean something like: "this user running this program on host with that IP address".

The basic idea behind authentication protocols is to put the party claiming to be A under a test, which only the real A could pass. This is called a *challenge-response* scheme, and can be implemented using either secret-key or public-key cryptographic methods.

Challenge-response using secret-key. Here, it is assumed that A and B share a secret key K.

- $A \rightarrow B$: "I am A".
- B : generates a random number x (challenge).
- $B \rightarrow A : x$.
- $A \rightarrow B : K(x)$.

B knows that only A knows K, so only A could have computed and sent $K(x)$.

Notice that when such a protocol is used, an eavesdropper can read both the plaintext x and the ciphertext $K(x)$.

Challenge-response using public-key. A first possibility is the following protocol:

- $A \rightarrow B$: "I am A".
- B : generates a random number x (challenge).

- $B \to A : x$.
- $A \to B : PrivK_A(x)$.

which is exactly like the one using secret keys, above. B knows that only A knows $PrivK_A$, so only A could have computed and sent $PrivK_A(x)$. This is sometimes called a *digital signature*: A "signs" x with its private key.

Another possibility is the "dual" protocol:

- $A \to B$: "I am A".
- B : generates a random number x (challenge).
- $B \to A : PubK_A(x)$.
- $A : PrivK_A(PubK_A(x)) = x$.
- $A \to B : x$.

Only A knows $PrivK_A$, so only A can find x from $PubK_A(x)$.

Nonces and randomness. The random number x used in the challenge is sometimes called a *nonce*. It is essential that the way nonces are generated is non-predictable, otherwise the above authentication mechanisms do not work. For example, if x was implemented by a counter, incremented by one each time B receives a new authentication request, then an intruder C could easily fool B. Consider for instance the last of the above protocols. First, C would observe the responses exchanged by B and the parties authenticating themselves to B. For instance, it would see $A \to B : x$, $A' \to B : x+1$, $A'' \to B : x+2$. Then, it would send a message to B pretending to be, say, D. B would reply with $PrivK_D(x+3)$. C, knowing the next nonce would be $x+3$, it would reply to B by sending $x+3$ and B would believe this indeed is D. The other protocols present similar problems (C could replay $PrivK_A(x)$ to some other node B', pretending to be A, when it realizes that the counter of B' has reached x).

Can authentication ruin your privacy? It turns out that the above protocols present serious problems, not really with respect to the requirement they are designed for (authentication) but with respect to affecting dangerously other parts of the security system. For example, the first of the public-key protocols above can be used in an insecure way by a malicious user B as follows. B has observed the conversation between A and C and has saved a message $PubK_A(M)$ exchanged between them. B would like to find what M is. The next time A asks to authenticate himself to B, B sends $x = PubK_A(M)$. Unsuspicious, A would compute and send back $PrivK_A(PubK_A(M)) = M$!

The second of the public-key protocols above can also be used maliciously by B. B wants to prove to someone else, say C, that he is A. When A asks to authenticate himself to B, B has his chance. B immediately asks to authenticate himself to C, pretending to be A. C sends back to B a challenge $PubK_A(x)$. B sends the same challenge to A, who replies with $PrivK_A(PubK_A(x)) = x$. B can now send x to C !

The second example is serious, so that the protocol should not be used for authentication. In the first example, one solution would be not to use the same set of keys both for privacy and authentication. That is, A has a set of keys $(PrivK_A, PubK_A)$ for signing challenges (authentication) and another set of keys $(PrivK'_A, PubK'_A)$ for encrypting/decrypting messages (privacy).

An example of a real security system offering, among others, authentication, is Kerberos (see below).

5 Integrity

Encryption of messages does not guarantee their integrity in general. For example, assume that A sends a message $PubK_B(M)$ to B. An intruder C could remove this message and send instead $PubK_B(M')$ to B (recall that C knows $PubK_B$). B would believe that A sent M'. Even using secret keys, integrity is not guaranteed. When A sends $K_{A,B}(M)$ to B, an intruder can remove it and send X instead. B will think that A sent $K_{A,B}(X)$. Notice that $K_{A,B}(X)$ might be garbage (e.g., not a sensible English phrase). However, we cannot base our integrity mechanism on the ability to distinguish valid messages from garbage, since there is no definition of what is "valid" and what is "garbage". Although this message might not make sense if we are talking about English text, it might cause real trouble in B, if it is part of numerical data used for computing a result in B, or some executable code, and so on. In any case, we would like B to know that A has not sent M', or $K_{A,B}(X)$, and discard the messages.

To ensure integrity, hashing functions are used. A hashing function H maps a large set of inputs to a small set of outputs. That is, H can map two different inputs to the same output.

To illustrate the idea of how H can be used for integrity, let us assume initially that H is known only by A and B. Then, when A wants to send M to B (M can be in-clear if privacy is not required, or encrypted if privacy is required) it sends the pair $(M, H(M))$. B receives some (possibly altered) message (X, Y) and checks whether $Y = H(X)$. If so, B believes that the message has not been altered.

For the above to work, a property of the function H must be the following: knowing M and $H(M)$, it is very hard to find M' such that $H(M') = H(M)$ (otherwise, an intruder could replace $(M, H(M))$ by $(M', H(M))$, even if the intruder does not know H).

In practice, we cannot require H to be secret, for the same reasons that the encryption/decryption functions cannot be secret. Therefore, hashing is used in combination with cryptography, to ensure integrity.

Combining hashing with secret-key cryptography. Assuming that K is shared only between A and B, A can send to B: $(M, H(M, K))$. This way, it is very hard for an intruder C to change M into M': C would have to find M' and Y such that $H(M', K) = Y$. Since C does not know K, it cannot find such M' and Y.

One might ask "why is K used in such a strange way? why not simply send $(M, K(M))$ if we want to protect the integrity of M?" In fact, this is possible: it is the same as assuming H to be secret, since K is secret. The reason not to use it is that computing $K(M)$ (for a large message M) can be much more expensive than computing $H(M, K)$. Also, $K(M)$ is as long as M, while $H(M, K)$ is much shorter, which reduces the size of the transmitted message.

Combining hashing with public-key cryptography. Here, A cannot send $(M, H(M, PrivK_A))$, since B does not know $PrivK_A$, thus cannot check the equality above. Instead A sends $(M, PrivK_A(H(M)))$. B, receiving (X,Y), first computes $Z = PubK_A(Y)$ and then checks whether $Z = H(X)$, in which case it believes the message has not been altered. In order for an intruder C to alter M, C would have to find M' such that $PrivK_A(H(M'))$. Not knowing $PrivK_A$, this is very hard. Alternatively, C could try to find M' such that $H(M') = H(M)$. We have assumed this to be very hard as well.

As previously, one may wonder why A needs to combine encryption and hashing, instead of just sending: $(M, PrivK_A(M))$, and letting B verify that: $M = PubK_A(PrivK_A(M))$. Economic reasons also apply here, but in this case there is a more serious problem. An intruder could remove the above message, and send instead: $(PubK_A(X), X)$, where X is some arbitrary message, different than $PrivK_A(M)$. The altered message would pass the test in B, and B would think that A sent $PubK_A(X)$ (which might be garbage).

Other combinations of hashing and cryptography to ensure integrity exist as well. One of them, appearing in [11], consists in A first generating a secret key K, then sending to B the triple: $(M, H(M,K), PrivK_A(K))$. The intention is that B first recovers K (knowing $PubK_A$), and then checks whether $M = H(M,K)$. However, this protocol is wrong: C (knowing $PubK_A$) could easily recover K from $PrivK_A(K)$; then C could replace M by M' and (knowing H and K) $H(M,K)$ by $H(M',K)$ and send finally: $(M', H(M',K), PrivK_A(K))$.

An example of hashing used in cryptography is the *message-digest* methods, the most common being the so-called MD5 [9].

6 Examples of Security Systems

So far, we have seen security mechanisms and protocols in isolation. It turns out that it is not obvious to build a complete security system by combining these mechanisms (we have even seen examples where one mechanism might be used in order to compromise another). Here, we present examples of complete real-life security systems. Since these systems are quite complicated, we limit ourselves to a simplified description. We also mention emerging protocols such as TLS or IPSEC.

6.1 Secure file-transfer or e-mail: PEM and PGP

Suppose A wants to send a message M (an e-mail, or a file) to B, ensuring the integrity and privacy of M. Also, B wants to make sure that M comes from A and not someone else.

The following protocol achieves this:

- A : generates random secret key K (e.g., 56-bit long).
- A : computes $E = K(M)$ (e.g., using DES).
- A : contacts CA to get $PubK_B$ and then computes $F = PubK_B(K)$ (using, for example RSA).
- A : computes $I = H(E,F)$ (where H is, for example, MD5).
- A : computes $G = PrivK_A(I)$.

- $A \to B : (E, F, G)$.
- B : receives (possibly altered) (E', F', G').
- B : computes $K' = PrivK_B(F')$.
- B : computes $M' = K'(E')$.
- B : contacts CA to get $PubK_A$ and then computes $I' = PubK_A(E')$.
- B : checks whether $I' = H(E', F')$.

The idea is the following. A generates a secret key K and uses it to encrypt the message M. A sends K to B encrypted using the public key of B. A protects the integrity of K and M by hashing and encrypting using its own private key. B first recovers K using its private key. B then decrypts M using K. Finally, B checks the integrity of the message and authenticates A at the same time, by decrypting the hashed pair and checking whether the hashed values match.

If the final test succeeds, B can be sure that M came from A: otherwise, I' would be different from $H(E', F')$, since only A knows $PrivK_A$. Also, B knows that M has not been altered: otherwise, I' would again be different from $H(E', F')$. Finally, B knows that nobody else knows M: to decrypt E, one would need K.

If the test fails, then either the message has been altered (i.e., $E' \neq E$ or $F' \neq F$ or $G' \neq G$), or the sender was not A in the first place (i.e., $E' = E$ or $F' = F$ or $G' = G$, but $PubK_A(G) \neq H(E, F)$, which implies that $G \neq PrivK_A(H(E, F))$, therefore, the sender didn't know $PrivK_A$).

Finally, A can also be certain that only B can read M, since only B has $PrivK_B$ and can thus decrypt K and then M.

The above protocol is the essence of PEM (privacy enhanced mail) and PGP (pretty good privacy). The difference between PEM and PGP is on how the certificates for public keys are obtained.

6.2 Secure remote-login and shell: Kerberos

Kerberos is based on the so-called Needham-Schroeder protocol. There are currently two version of Kerberos, V4 and V5, the first one of which is more specific to systems running the TCP/IP stack, whereas V5 is more general. The main objective of Kerberos is to provide to the user remote access to resources (i.e., access over the network) while at the same time ensuring that the user has the right to access these resources. In other words, Kerberos can be viewed as a "login" protocol, which is secure even if the login is remote (classic rlogin is not secure, since the password travels in the network).

The idea behind Kerberos is to use a trusted third party, which we call S. S knows the passwords of all users, and can compute a secret key K_A (e.g., a DES key) from the password P_A of each user A. That is, S shares the secret key K_A with A.

The protocol goes as follows:

- (Message 1) $A \to S$: "I am A, I want to talk to B".
- S : computes a random session key K, to be used by A and B

- S : computes a timestamp T_S (the current time at S).
- (Message 2) $S \to A : K_A(T_S, L, K, B)$.
- (Message 3) $S \to A : K_B(T_S, L, K, A)$ (this is called the *ticket*).
- A : decrypts $K_A(K_A(T_S, L, K, B)) = (T_S, L, K, B)$; if the last field is not B, then the message from S has been altered and A stops.
- (Message 4) $A \to B : K_B(T_S, L, K, A)$.
- B : decrypts $K_B(K_B(T_S, L, K, A)) = (T_S, L, K, A)$; if the last field is not A, or if the current time at B is greater than $T_S + L$, then B stops.
- A : generates a nonce T_A (does not have to be random in this case, in fact it is usually the current time of A) and then computes $K(A, T_A)$ (this is called the *authenticator*).
- (Message 5) $A \to B : K(A, T_A)$.
- B : decrypts $K(K(A, T_A)) = (A, T_A)$; if the first field is not A, then B stops.
- (Message 6) $B \to A : K(T_A + 1)$.
- A : compute $K(K(T_A + 1))$ and check whether it is indeed equal to $T_A + 1$.

The idea of the protocol is the following. S gives to A a fresh key K, which A recovers using its password K_A. This key will be used for *mutual* authentication of A and B. It will also be used during the communication of A and B. S also gives to A a ticket that proves that A has the right to access the resources in B (in fact, B can itself be a resource, such as a remote printer, disk, etc). A cannot decrypt or alter this ticket, it can only pass it to B.

When B receives the ticket from A, it decrypts it using its password K_B. If B sees that the ticket is too old or does not belong to A as claimed, then B stops the transaction (i.e., does not send to or accept any more messages from A). The value L is the *lifetime* of the ticket, i.e., it specifies for how long the ticket and the session key K are valid. We don't want the key to be valid for a long time, since someone could break it meanwhile. Also, we don't want the key to be valid just once, since it wouldn't be efficient if A had to get a new ticket from S and send it to B each time A wanted to perform an operation, such as `ls -l`, on B. L is typically in the order of 10 hours. If all goes OK, at this point B believes that A has the right to use a resource, and that A knows the session key K.

A also sends an authenticator to B. The authenticator serves two purposes. First, it proves to B that A knows K (the ticket does not prove that, since C could have masqueraded as A and asked S for the ticket). Second, the authenticator contains a challenge for B. That is, not only B wants to authenticate A, but also A wants to authenticate B. The challenge is for B to prove knowledge of K, therefore, knowledge of K_B (since B cannot recover K otherwise). If the test after message 6 succeeds, A believes that B knows K. [1]

[1] One might wonder why B sends $K(T_A + 1)$ instead of $K(T_A)$. This is probably because the designers of Kerberos anticipated that encryption algorithms such as DES will be used to compute $K(\cdot)$. In these algorithms, a large message is encoded block-by-block. Sometimes (although in recent versions this is no longer true) this block encryption has the property: $K(M_1, M_2) = (K(M_1), K(M_2))$. In our case, this could give $K(A, T_A) = (K(A), K(T_A))$. Therefore, B could simply return the second part of the message sent by A, namely, $K(T_A)$. Requiring B to send $K(T_A + 1)$ removes this possibility.

At the end of the above exchange of messages, A and B have mutually authenticated themselves, and they share a secret session key K. B can now accept messages (e.g., shell commands) from A encrypted with K, and can return the results also encrypted with K. The key is valid only for L time units. If A wants to continue talking to B for more than L, it has to contact S and get a new ticket (with a new session key K').

In reality, there are more than one tickets involved in a typical Kerberos session. First, A has to get a *ticket-granting* ticket, that will allow him to use the so-called "services" on B (e.g., check e-mail on B, print a file from B, etc). Each time A wants to use a new service (for which it does not have a ticket yet, or for which the ticket has expired), it uses the (key contained in the) ticket-granting ticket, in order to get a new *service* ticket. Then it uses the (key contained in the) service ticket to access the service.

[1] mentions some problems of Kerberos with the use of timestamps, and gives references to other related papers.

6.3 SSL, TLS and HTTPS

The secure socket layer (SSL) is a set of security mechanisms implemented as a layer between TCP and the application layer. It allows applications to open secure TCP connections, instead of plain TCP connections. A secure connection can mean various things. Authentication might be required, privacy, integrity, some, or all of them. Moreover, it is possible to negotiate the used algorithms (whether RSA or DES, how long keys, etc) upon connection establishment.

TLS (transport layer security) is the corresponding upcoming IETF standard, based on SSL. When HTTP is used on top of SSL (or TLS) it is called HTTPS (secure HTTP). In fact, HTTP does not really change: instead of opening connections, it opens secure TCP connections.

6.4 Security at the network layer: IPSEC

IPSEC [4] aims at providing security at the network level (e.g., implemented in routers). The motivation is that some security requirements are not specified per connection, but instead for the whole aggregate traffic flowing between two networks. For example, a corporation that has two remote sites (networks) might want privacy for all e-mail messages exchanged between these sites. Instead of achieving this privacy by using security protocols for each e-mail, a more efficient way might be to dictate to a set of routers that all IP packets with this source and that destination network must be encrypted somehow.

6.5 Firewalls

Firewalls do not provide security directly in terms of the three requirements we set and used throughout these notes. Instead, they mainly protect from *denial-of-service* attacks, where a malicious (or careless) user floods a network with garbage packets, preventing others from doing their work.

A firewall is a router configured to filter (i.e., drop) incoming or outgoing packets that match a specified pattern. For example, a firewall might be the only access point of a private corporation network to the rest of the Internet. This firewall might implement a filter specifying that all incoming packets with destination IP address other than a specific host should be dropped. Firewalls might also implement filters based on the source and/or destination port numbers, in which case the IP layer running at the router needs to look at the TCP header of the packets.

References

[1] M. Abadi and R. Needham. "Prudent Engineering Practice for Cryptographic Protocols", IEEE Transactions on Software Engineering, Jan. 1996.

[2] C. Kaufman, R. Perlman and M. Speciner. "Network security: private communication in a public world", Prentice-Hall, 1995.

[3] Cryptography frequently asked questions. `http://www.faqs.org/faqs/cryptography-faq/`.

[4] IPSEC RFC 1825, 1826 and 1827.

[5] K. Jackson and J. Hruska (editors). "Computer Security Reference Book", CRC Press, 1992.

[6] Kerberos frequently asked questions. `http://www.faqs.org/faqs/kerberos-faq/general/index.html`.

[7] Kerberos RFC 1510.

[8] MD4 RFC 1320.

[9] MD5 RFC 1321.

[10] Snake-oil frequently asked questions. `http://www.faqs.org/faqs/cryptography-faq/snake-oil/index.html`.

[11] L. Peterson and B. Davie. "Computer networks: a systems approach", Morgan-Kaufmann, 1996. Also 2nd edition, 1999.

[12] J. Walrand. "Communication networks: a first course", 2nd edition, McGraw-Hill, 1998.