Security

Goals and Threats:

1. data confidentiality: being able to read data
2. data integrity: being able to modify data
3. system availability: denial of service

Cryptography

Sub-topic is cryptography.

Cryptography has its roots in the military, where low-level clerks (who could not be trusted) carried secret messages between commanders. The problem there was exchanging secret messages between high-level officers, when those messages were carried across hostile territories by low-level (possibly untrustable) clerks, with the inherent possibility that messages might fall into enemy hands. Today:

- Networks are the "low-level clerks"because most network technologies can be easily tapped. That is, they cannot be trusted. LANs such as Ethernets, for instance, allow any station to easily examine all packets transmitted between any two machines. Likewise, WANs often use the phone systems, and some companies (e.g, competitors) will not trust them.
- While most users don't require secrecy in the military sense, most users don't want third parties to intercept and read their private mail.

In networks, several types of problems:

Secrecy: Keeping information out of unauthorized users hands.

Authentication: Problem of *impersonation* where an intruder might masquerade as a legitimate user. Such as obtaining message copies through *wiretapping* and then using a *replay attack*. An impersonator could ask for information that they are not authorized to see, or generate messages causing incorrect (and certainly unauthorized) transactions to take place (e.g., transferring money from one account to another). Note that an impersonator could intercept and modify messages, or create new ones.

Nonrepudiation: Digitial signatures so a user cannot disclaim an agreement.

Integrity Control : Message was the one really sent.

The only general solution to security issues is to *encrypt* a user's *plaintext* messages into *ciphertext* messages. Ciphertext is a "jumbled" form of the message that can only be used if converted back into its original cleartext. It is the ciphertext that is sent across the network, and the intended recipient *decrypts* the ciphertext back into the original plaintext.

One of the fundamental rules of computer security is that an intruder knows a lot, if not everything, about the security system in use. "Security through obscurity" does not work! If a system has a vulnerability, one must assume that the intruder is aware of that vulnerability.

If encryption is used, for instance, one should assume that the intruder has a general idea of the how the encryption algorithm works.

The art of breaking ciphers is called *cryptanalysis*. The art of devising ciphers is called *cryptography*. Cryptology refers collectively to the task of making and breaking ciphers.

One problem with encryption is that once an intruder has determined the encryption algorithm, it must be changed immediately. How? Changing an algorithm may be impractical.

A common solution is to break the encryption process into two parts: Use a general encryption algorithm that uses a (short) string of characters called a *key* to select one of many encryptions to use. Now the code for the encryption algorithm changes rarely, but the key can be changed as often as needed.

Functions are used to encrypt and decrypt messages:

$Ciphertext = E_K(Plaintext)$ and

$Plaintext = D_K(Ciphertext)$; In the above terminology, E_K means the message is encrypted using a key of K, while D_K denotes the decryption function using a key of K.

The properties of encryption functions depend on the specific function in use. For example, some functions are designed in such a way that the same key encrypts and decrypts, but that the encryption and decryption functions differ. For other functions, the same function may perform both encryption and decryption (e.g., encrypt and decrypt are inverses of each other).

From a cryptanalyst's point of view, the immediate goal is to guess the keys being used in so that messages can be decrypted (or bogus messages can be encrypted). The cryptanalysis problem has four variations:

Ciphertext Only: Ciphertext but no plaintext is present. Given only the ciphertext, determine the key and the original text. The ciphertext only problem is particularly difficult to solve because we may not even know when we have guessed the correct key because we don't know what the unencrypted data looks like.

Verifiable Plaintext: We don't have any actual plaintext, but given ciphertext, we can verify that we have guessed the correct key. This is weaker than the above, in that even though we don't have plaintext, we can (somehow) determine whether or not we have guessed the correct key. Example? A pair of (encrypted) request-response messages may carry the same sequence number, so that the client can match response messages with the original request. When decrypting the messages, we know we have guessed the correct key if we obtain the same sequence number when decrypting both messages. Thus, although we don't have any plaintext, we can verify that we have obtained the plaintext from the ciphertext. Another example is decrypting messages that we know contain ascii text. We have know idea what the text is, but we can certainly recognize whether its ascii or not.

Known Plaintext: Both plaintext and its encrypted ciphertext are available. In this case, verifying that we have guessed the key is straightforward, since we can test our key on the plaintext.

Chosen Plaintext: Here, we have the ability to encrypt pieces of plaintext of our own choosing.

How are encryption algorithms broken? Brute force. We can always try all possible keys, in an attempt to find the one that works. The amount of effort needed to break encryption algorithms using brute force depends on two factors: 1) the amount of time needed to encrypt (decrypt) a message using a given key, and 2) the total number of possible keys that must be tried in the worst case. If it takes a long time (days to years) to try all keys, it is unlikely that anyone can guess a key using brute force.

In practice, brute force is not always needed. Why? Brute force is needed only if all combinations of bits (in keys) are equally likely to be present. If keys are user-chosen passwords, then it may well be a word in a dictionary. In addition, most messages contain patterns. The more messages one has available, the easier it becomes to recognize patterns. If messages contain english text, for instance, certain letters and phrases appear more often than others. Moreover, if the same key is used to encrypt subsequent messages, patterns in the original message will translate into patterns in the ciphertext. Recognizing patterns reduces greatly the number of combinations that need to be tried in guessing original plaintext.

Note: The ciphertext only problem is the most difficult to solve. However, it is naive to assume that a cipher that can withstand a ciphertext only attack is secure. In many cases, a cryptanalyst can make educated guesses as to some the plaintext. For instance, remote login sessions usually start with a message such as "Please Login:". Thus, most present-day commercial and government agencies desire that a cryptosystem be able to withstand a chosen plaintext attack.

One-Time Pad

How can we prevent statistical attacks? Never reuse the encryption key.

Unfortunately, all current encryption methods have problems. Consider the *one-time pad*:

1. It uses a *random key* that has the same length as the message being encrypted. Both the encryptor and decryptor use the same key.
2. The message is exclusive-or'd with the key to produce the ciphertext.
3. The receiver decrypts the ciphertext using the same algorithm and key. ((A XOR K) XOR K) = A.
4. Have we solved our original problem? No! Because the sender and receive use the same key, the key must be transferred to the receiver. Exchanging keys is the very same problem we are trying to solve!
5. The key can never be reused (or it becomes easier to guess). That is, reusing keys allows statistical based attacks on the key.
6. Choosing keys presents difficulties; in particular, "randomly generated" keys are not very random at all. That is, computer generated random numbers are actually well defined sequences. Although the generated numbers are uniformly distributed, they are not random.
7. It is the only *provably secure* method known at this time.

Data Encryption Standard (DES)

The *data encryption standard* (DES), developed by the NBS, has become one popular encryption method:

1. The calculation can be performed efficiently using special hardware, but far less efficiently by normal programs.
2. Keys are 56 bits in size, large enough that they are difficult to guess.

However, critics argue that both of these assumptions are false given today's technology. In particular:

1. Are 56-bit keys too small? Many people say yes. Indeed, IBM's original design used 128-bit keys.
2. Why aren't keys bigger? The key size was reduced at the request of the National Security Agency (NSA), who has never given a reason for the change.
3. IBM has never given a technical reason for the specific design of its algorithm. Why? Again, this was done at NSA's request.
4. Without knowing the design principles, it is difficult to know whether the code can be broken easily. Indeed, some critics suggest that the NSA would feel very uncomfortable with an encryption standard that not even they can break.

Both DES and one-time pad are considered *conventional* methods because they use the same key for both encryption and decryption. Conventional methods have the problems that keys must be distributed to both the sender and recipient.

Public Key Cryptography

Until now, we have assumed that hiding keys is of the utmost importance. However, this leads to the key distribution problem. Another approach, called Public Key Encryption, uses two keys: a *public* key and a *private* key. The public key is given to everyone, while the private key is kept secret. It requires:

1. $Plaintext = Decrypt(Encrypt(Plaintext))$. (E.g., the encrypt and decrypt operations are inverses of each other.)
2. It must be extremely difficult to deduce *Decrypt* given *Encrypt*. That is, the encryption function is made available to everyone, so it shouldn't be possible for someone to deduce the decryption function given the encryption function.
3. *Decrypt* cannot be broken by a chosen plaintext attack. Since everyone has the encryption function, anyone can present their own chosen plaintext to the encryption algorithm.

Under rules 2 and 3, *Encrypt* can be made public and distributed to everyone else. *Decrypt* remains private.

The RSA (Rivest, Shamir, Adleman) algorithm is an encryption algorithm that satisfies the above requirements. It is based on the factoring of large prime numbers, which no one knows how to do efficiently. Thus, RSA encryption is considered quite strong. RSA encryption uses a *public* and *private* key. The public key is made available to everyone, but the private key is kept secret. RSA encryption takes place as follows:

$Plaintext = K_{pri}(K_{pub}(Plaintext))$, where K_{pub} denotes encryption using K's public key, and K_{pri} denotes encryption using K's private key.

Security

Security implies that only the recipient of a message may decrypt it.

Now, when A wants to send a message to B, A encrypts messages for B using B's public key and sends the cipher to B, which B decrypts with its private key.

Can anyone else decrypt messages intended for B? No, one must have B's private key to decrypt the message, which only B has.

Unfortunately, although only B can decrypt the message, B cannot be sure that A actually sent the message.

$cipher = B_{pub}(Msg)$, $msg = B_{pri}(cipher)$

Authentication

Authentication is concerned with verifying that a message supposedly from B, actually did originate from B.

The key to authentication is having client A provide information that only A knows. Examples? For example, a password. The drawback of a password is an intruder might see the password (if it is unencrypted) and would then be able to gain unauthorized access to the system. Even better would be the encryption of a number using a key that only A knows.

For authentication, it is convenient to assume that:

$$Message = K_{pri}(K_{pub}(Message)) = K_{pub}(K_{pri}(Message))$$

That is, that a user's decryption key can also encrypt messages. The RSA algorithm discussed above has this property.

How does this give us authentication? Assume A wants to authenticate itself to B:

1. B selects a random number X and sends $A_{pub}(X)$ to A.
2. A computes $A_{pri}(A_{pub}(X)) = X$. That is, it decrypts the received message using its private key.
3. A returns $A_{pri}(X + 1)$, which B decrypts. B knows it is communicating with A if it gets the expected answer of X+1. That is, to compute $X + 1$, A must have been able to decrypt X, which means it must have A's private key.

Problem? We still do not yet have secrecy. Although B can verify that a message came from A, anyone else who obtains a copy of the message can decrypt it too! We can combine the two ideas to create messages that are both secret and authenticated. A sends the following message to B:

$$Cipher = A_{pri}(B_{pub}(Message))$$

which B processes via:

$$B_{pri}(A_{pub}(Cipher)) = Message$$

Note:

1. B is sure that A sent the message because only A knows A_{pri}. This is how we get authentication — forcing A to produce some information that only A can have.
2. A is sure that only B can read the message because only B knows its private key (secure)
3. Unfortunately, each message transaction involves four iterations of the functions, adding substantial cost.

Replay Attacks

Protection against replay attacks (where someone replays old message in the hopes of forcing a server to (improperly) re-perform a transaction) can be handled as follows:

1. Client sends request to server.
2. Server returns to client a random number R (in an encrypted message).
3. Client returns $(R + 1)$ to server.
4. Have each transaction uses a different random number. This effectively prevents replays because the returned $(R + 1)$ will only match the old transaction, not the one being performed now. That is, if the same transaction were done twice (legitimately), different transaction numbers would be used. Thus, replay attacks are not possible.

The key here is that the server forces the client to prove that it knows its private key before doing a transaction. Thus, and intruder cannot record a transaction and play it back later — the server would generate a new random number that the client could not decrypt.

Digital Signatures

In the real world, signatures are legally binding. If someone signs a contract, the signature is proof that a party did agree to the terms of the contract. How can we have the equivalent of signatures with computers?

The problem of sending a "signed" message from one party to another has two parts:

Authentication: Having the receiver verify the claimed identity of the sender (e.g., "Are you really cew@cs.wpi.edu?"). We have already discussed how authentication can be done.

Digital signature: Preventing the sender from later repudiating the message (e.g., "Hey, I never said that!").

Digital signatures can be achieved by using the public key encryption. Suppose B is concerned that A will later disavow having sent a message:

1. A sends to B: $B_{pub}(A_{pri}(Message))$.
2. B decrypts the message using its private key, but saves the *signature* $A_{pri}(Message)$ in archival storage.
3. B then decrypts the message using A's public key and processes the transaction.

Later, when A denies ever having sent the message, B produces *signature*. Why can't A disavow the signature? Only A's private key could have produced it.

User-Authentication

The problem of identifying users when they log in is called *user authentication*. Most authentication methods are based on:

1. what the user knows (e.g., passwords)
2. what the user has (e.g., a plastic card)
3. physical characteristics of the user (e.g., fingerprints)

Passwords

The most widespread form of authentication is requiring the user to enter a *password*.

In the first implementation of Unix, a password file contained the actual password for each user. This approach had several problems:

1. there is no way to prevent the making of copies by privileged users
2. software (or human) errors could cause the contents of the file to become available to others
3. the contents of the file saved on backup tapes were available to anyone with physical access to the tapes

Another possible approach is to encrypt each user's password with some key and store the encrypted version:

$$Encypted\_Password = Encrypt(Password,\ Key)$$

Now, when the user tries to log in, his password is encrypted and compared with the encrypted version in the password file. If the two match, the log in succeeds.

Moreover, if the encryption function is hard to invert (even with a key), then the password file could be read by any programs. Encryption functions that are hard to invert are called *trap-door* functions.

The next version of the password encryption used an algorithm that simulated the M-209 cipher machine used by the US Army during World War II. Because the cyphertext it produced was easily invertible (given the key), the password was used as a key to encrypt a constant.

Attacks on Encrypted Passwords

One approach to penetrating this scheme is to keep guessing the key until one succeeds. Brute force and fast systems.

The search can be further speeded up by first trying:

1. the 250,000 words in a dictionary (spelled forwards and backwards)
2. list of first names, last names, street names, and city names
3. valid license plates in the state
4. room numbers, social security numbers, telephone numbers, etc.

Morris and Thompson (1979), authors of the Unix password scheme, compiled a list of likely passwords using the above heuristics, encrypted each of them using the known encryption algorithm, and compared them with a list of encrypted password available at their site.

Over 86% of all passwords turned up in their list!

Now systems impose requirements on passwords to ensure users use a variety of characters in passwords.

Salted Passwords

Consider an intruder attempting to gain access to as many accounts on as many systems as possible. For each encrypted password he precomputes (from a list of good candidates of course), he can check the entries for all users.

The technique of *salted passwords* renders such attacks useless. Unix modifies the previous algorithm as follows:

1. when a new password is being entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends it to the password entered by the user.
2. the concatenated string of the 12-bit *salt* and the first eight characters of the password are used as a key
3. both the encrypted password and the 12-bit salt are stored in the password file
4. when the user subsequently attempts to log in, the 12-bit salt is extracted from the file and combined with the typed password

Now an intruder can no longer amortize the cost of one encryption over all the password entries to be searched.

Other Methods

In password protected systems, the main idea is that authorized users have a certain piece of information that they present to the system. A generalization of this idea is to have the computer keep a large amount of information that only an authorized user knows. Rather than ask for a password, the system can ask the user a series of questions:

- What is your quest?
- What is your favorite color?
- How fast does a sea gull fly?

At login time, the system asks a series of questions (chosen at random) that the user is expected to know. Because the set of questions changes for each login attempt, an intruder can't gain access by looking over a user's shoulder when he enters his password.

Other Approaches

Smart card for authentication (Fig 9-7). Use smart card at client site to determine response to a challenge from remote computer.

Biometrics—fingerprint, voice.

Attacks from Inside the System

So what can be done once access is obtained (either legitimately or from an intruder)?

Relevant to OS study.

Trojan horse (substitute a malicious copy) of a well-known program

1. leave a phony program to simulate login program
2. place a program with a well-known name in the user's path so the user unknowingly executes the non-standard copy of a program. A reason not to include "." in command path. The non-standard copy can emulate normal command plus have user permissions for access to other objects.
3. logic bombs left in code that activate based on a particular time or event (or non-event if they must be neutralized).
4. trap doors to allow access with a special password or login.
5. buffer overflow attacks exploit knowledge of code structure to possibly gain access privileges of a program that is attacked.

Attacks from Outside the System

Viruses can be spread when programs execute code sent to them—such as email attachments.

Try to limit the available operations—sandboxing—such as Java applets.

Work to limit the capabilities of executing such attachments (SubOS work).

Anomaly detection—build up a profile of standard behavior and then look for anomalies in access patterns.

A virus may work by substituting copies of known programs—a trojan horse! Examine means to detect when non-standard system call sequences from standard programs are generated.

worm—self-replicating program.