INFORMATION SECURITY

Prepared by:- Prof. Sarbasri Sahoo

EINSTEIN ACADEMY OF TECHNOLOGY & MANAGEMENT
(Managed by Udayanath Educational & Charitable Trust, Bhubaneswar)

AICTE BPUT SCTE & VT www.eatm.in

Department of Computer Science and Engineering
Einstein Academy of Technology & Management
Baniatangi, Khordha, Odisha-752060

MODULE 1

The security problem in computing

1.1 The meaning of computer security

The meaning of the term computer security has evolved in recent years. Before the problem of data security became widely publicized in the media, most people's idea of computer security focused on the physical machine. Traditionally, computer facilities have been physically protected for three reasons:

• To prevent theft of or damage to the hardware

• To prevent theft of or damage to the information

• To prevent disruption of service

Computer security is security applied to computing devices such as computers and smartphones, as well as computer network such as private and public networks, including the whole Internet. The field covers all the processes and mechanisms by which digital equipment, information and services are protected from unintended or unauthorized access, change or destruction, and are of growing importance in line with the increasing reliance on computer systems of most societies worldwide. It includes physical security to prevent theft of equipment, and information security to protect the data on that equipment. It is sometimes referred to as "cyber security" or "IT security", though these terms generally do not refer to physical security (locks and such).

Some important terms used in computer security are:

Vulnerability

Vulnerability is a weakness which allows an attacker to reduce a system's information assurance. Vulnerability is the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw. To exploit vulnerability, an attacker must have at least one applicable tool or technique that can connect to a system weakness. In this frame, vulnerability is also known as the attack surface.

Vulnerability management is the cyclical practice of identifying, classifying, remediating, and mitigating vulnerabilities. This practice generally refers to software vulnerabilities in computing systems.

Backdoors

A backdoor in a computer system, is a method of bypassing normal authentication, securing remote access to a computer, obtaining access to plaintext, and so on, while attempting to remain undetected.

The backdoor may take the form of an installed program (e.g., Back Orifice), or could be a modification to an existing program or hardware device. It may also fake information about disk and memory usage.

Denial-of-service attack

Unlike other exploits, denials of service attacks are not used to gain unauthorized access or control of a system. They are instead designed to render it unusable. Attackers can deny service to individual victims, such as by deliberately entering a wrong password enough consecutive times to cause the victim account to be locked, or they may overload the capabilities of a machine or network and block all users at once. These types of attack are, in practice, very hard to prevent, because the behavior of whole networks needs to be analyzed, not only the behavior of small pieces of code. Distributed denial of service (DDoS) attacks are common, where a large number of compromised hosts (commonly referred to as "zombie computers", used as part of a botnet with, for example; a worm, trojan horse, or backdoor exploit to control them) are used to flood a target system with network requests, thus attempting to render it unusable through resource exhaustion.

Direct-access attacks

An unauthorized user gaining physical access to a computer (or part thereof) can perform many functions, install different types of devices to compromise security, including operating system modifications, software worms, key loggers, and covert listening devices. The attacker can also easily download large quantities of data onto backup media, for instance CD-R/DVD-R, tape; or portable devices such as key drives, digital cameras or digital audio players. Another common technique is to boot an operating system contained on a CD-ROM or other bootable media and read the data from the hard drive(s) this way. The only way to defeat this is to encrypt the storage media and store the key separate from the system. Direct-access attacks are the only type of threat to Standalone computers (never connect to internet), in most cases.

Eavesdropping

Eavesdropping is the act of surreptitiously listening to a private conversation, typically between hosts on a network. For instance, programs such as Carnivore and Narus Insight have been used by the FBI and NSA to eavesdrop on the systems of internet service providers.

Spoofing

Spoofing of user identity describes a situation in which one person or program successfully masquerades as another by falsifying data and thereby gaining an illegitimate advantage.

Tampering

Tampering describes an intentional modification of products in a way that would make them harmful to the consumer.

Repudiation

Repudiation describes a situation where the authenticity of a signature is being challenged.

Information disclosure

Information Disclosure (Privacy breach or Data leak) describes a situation where information, thought as secure, is released in an untrusted environment.

Elevation of privilege

Elevation of Privilege describes a situation where a person or a program want to gain elevated privileges or access to resources that are normally restricted to him/it.

Exploits

An exploit is a piece of software, a chunk of data, or sequence of commands that takes advantage of a software "bug" or "glitch" in order to cause unintended or unanticipated behaviors to occur on computer software, hardware, or something electronic (usually computerized). This frequently includes such things as gaining control of a computer system or allowing privilege escalation or a denial of service attack. The term "exploit" generally refers to small programs designed to take advantage of a software flaw that has been discovered, either remote or local. The code from the exploit program is frequently reused in Trojan horses and computer viruses.

Indirect attacks

An indirect attack is an attack launched by a third-party computer. By using someone else's computer to launch an attack, it becomes far more difficult to track down the actual attacker. There have also been cases where attackers took advantage of public anonymizing systems, such as the tor onion router system.

Computer crime: Computer crime refers to any crime that involves a computer and a network.

Top 10 Cyber Crime Prevention Tips

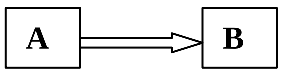
1. **Use Strong Passwords**
   Use different user ID / password combinations for different accounts and avoid writing them down. Make the passwords more complicated by combining letters, numbers, special characters (minimum 10 characters in total) and change them on a regular basis.
2. **Secure your computer**
   - **Activate your firewall**
     Firewalls are the first line of cyber defence; they block connections to unknown or bogus sites and will keep out some types of viruses and hackers.
   - **Use anti-virus/malware software**
     Prevent viruses from infecting your computer by installing and regularly updating anti-virus software.
   - **Block spyware attacks**
     Prevent spyware from infiltrating your computer by installing and updating anti-spyware software.

3. **Be Social-Media Savvy**
   Make sure your social networking profiles (e.g. Facebook, Twitter, YouTube, MSN, etc.) are set to private. Check your security settings. Be careful what information you post online. Once it is on the Internet, it is there forever!
4. **Secure your Mobile Devices**
   Be aware that your mobile device is vulnerable to viruses and hackers. Download applications from trusted sources.
5. **Install the latest operating system updates**
   Keep your applications and operating system (e.g. Windows, Mac, Linux) current with the latest system updates. Turn on automatic updates to prevent potential attacks on older software.
6. **Protect your Data**
   Use encryption for your most sensitive files such as tax returns or financial records, make regular back-ups of all your important data, and store it in another location.
7. **Secure your wireless network**
   Wi-Fi (wireless) networks at home are vulnerable to intrusion if they are not properly secured. Review and modify default settings. Public Wi-Fi, a.k.a. "Hot Spots", are also vulnerable. Avoid conducting financial or corporate transactions on these networks.
8. **Protect your e-identity**
   Be cautious when giving out personal information such as your name, address, phone number or financial information on the Internet. Make sure that websites are secure (e.g. when making online purchases) or that you've enabled privacy settings (e.g. when accessing/using social networking sites).
9. **Avoid being scammed**
   Always think before you click on a link or file of unknown origin. Don't feel pressured by any emails. Check the source of the message. When in doubt, verify the source. Never reply to emails that ask you to verify your information or confirm your user ID or password.
10. **Call the right person for help**
    Don't panic! If you are a victim, if you encounter illegal Internet content (e.g. child exploitation) or if you suspect a computer crime, identity theft or a commercial scam, report this to your local police. If you need help with maintenance or software installation on your computer, consult with your service provider or a certified computer technician.

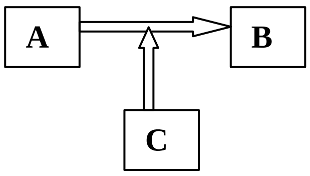
<u>Principle security</u>

There are five principles of security. They are as follows:

- *Confidentiality:*
  The principle of confidentiality specifies that only the sender and the intended recipient should be able to access the content of the message.



- *Integrity:*
  The confidential information sent by A to B which is accessed by C without the permission or knowledge of A and B.



- *Authentication:*
  Authentication mechanism helps in establishing proof of identification.

- Non-repudiation:

- *Access control:*
  Access control specifies and control who can access what.
- *Availability:*
  It means that assets are accessible to authorized parties at appropriate times.

Attacks

We want our security system to make sure that no data are disclosed to unauthorized parties.

- ➢ Data should not be modified in illegitimate ways
- ➢ Legitimate user can access the data

Types of attacks

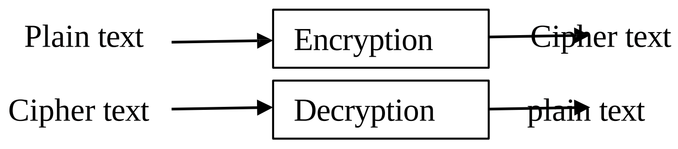
Attacks are grouped into two types:

- *Passive attacks:* does not involve any modification to the contents of an original message
- *Active attacks:* the contents of the original message are modified in some ways.

1.4 ELEMENTARY CRYPTOGRAPHY: SUBSTITUTION CIPHER

Encryption is the process of encoding a message so that its meaning is not obvious; decryption is the reverse process, transforming an encrypted message back into its normal, original form. Alternatively, the terms encode and decode or encipher and decipher are used instead of encrypt and decrypt. That is, we say that we encode, encrypt, or encipher the original message to hide its meaning. Then, we decode, decrypt, or decipher it to reveal the original message. A system for encryption and decryption is called a cryptosystem.

The original form of a message is known as plaintext, and the encrypted form is called cipher text. For convenience, we denote a plaintext message *P* as a sequence of individual characters *P* = <*p*1, *p*2, …, *pn*>. Similarly, cipher text is written as *C* = <*c*1, *c*2, …,*cm*>.



For instance, the plaintext message "I want cookies" can be denoted as the message string <*I, ,w,a,n,t,c,o,o,k,i,e,s*>. It can be transformed into cipher text<*c*1, *c*2, …,*c*14>, and the encryption algorithm tells us how the transformation is done.

We use this formal notation to describe the transformations between plaintext and cipher text. For example:
we write C = E (P) and P = D(C), where C represents the cipher text, E is the encryption rule, P is the plaintext, and D is the decryption rule.

P = D (E(P)).

In other words, we want to be able to convert the message to protect it from an intruder, but we also want to be able to get the original message back so that the receiver can read it properly. The cryptosystem involves a set of rules for how to encrypt the plaintext and how to decrypt the cipher text. The encryption and decryption rules, called algorithms, often use a device called a key, denoted by *K*, so that the resulting cipher text depends on the original plaintext message, the algorithm, and the key value. We write this dependence as *C* =*E* (*K*, *P*). Essentially, *E* is a *set* of encryption algorithms, and the key *K* selects one specific algorithm from the set.
There are many types of encryptions. In the next sections we look at two simple forms of encryption: substitutions in which one letter is exchanged for another and transpositions, in which the order of the letters is rearranged.

Cryptanalyst: cryptanalyst is a person who studies encryption and encrypted message and tries to find the hidden meanings (to break an encryption).

Confusion: it is a technique for ensuring that ciphertext has no clue about the original message.

Diffusion: it increases the redundancy of the plaintext by spreading it across rows and columns.

Substitutions Cipher: It basically consists of substituting every plaintext character for a different cipher text character.
It is of two types-

I. Mono alphabetic substitution cipher
II. Poly alphabetic substitution cipher

Mono alphabetic substitution cipher:
Relationship between cipher text symbol and plain text symbol is 1:1.

- Additive cipher:
  Key value is added to plain text and numeric value of key ranges from 0 – 25.

  Example:
  Plain text(P)- H E L LO (H=7,E=4,L=11,L=11,O=14)
  Key (K)=15
  Cipher text (C)= 7+15,4+15,11+15,11+15,14+15
  = 22,19, 26,26,(29%26)=3
  = W T A AD

- Affine cipher:
  It is the combination o tive cipher

$$\boxed{\begin{array}{l} C= (P+K) \bmod 26 \\ P= (C-K) \bmod 26 \end{array}}$$

Let K1 and K2 are two keys

$$\boxed{\begin{array}{l} C= [(P \times K1) + K2] \bmod \\ 26\ P= [(C-K2) \times K1^{-1}] \end{array}}$$

Polyalphabetic substitution cipher

In polyalphabetic cipher each occurrence of a character may have different substitution. The relationship between characters in plain text and cipher text is 1 to many.

- Auto key cipher
- Playfair cipher
- Vigegeire cipher
- Hill cipher

Auto key cipher:

- ➢ In this cipher, key is stream of subkeys in which subkey is used to encrypt the corresponding character in the plain text.
- ➢ Here 1st subkey is predefined and 2nd subkey is the value of the 1st character of the plain text 3rd subkey is the value of the 2nd plain text and so on.

Example: A T T A C K

0 19 19 0 2 10

Key=12 $\searrow \searrow \searrow \searrow \searrow$

12 0 19 19 0 2

Cipher text(C)= (12,19,38 19,2 12)%26 $\longrightarrow$ M T M T C M

Playfair cipher

In playfair cipher the secret key is made of 25 characters arranged in 5x5 matrix

Rules:-

- ➢ If 2 letters in a plaintext are located in the same row of the secret key then the corresponding encrypted character for each letter is next letter to the right.
- ➢ If 2 letters in a pair are in same column then the corresponding encrypted character is next below in the same column.
- ➢ If 2 letters are neither in same row or in same column then encrypted character is in its own row but in the same column as the other character.

Example:

$$K = \begin{vmatrix} L & G & D & B & A \\ Q & M & H & E & C \\ U & R & N & I/J & F \\ X & V & S & O & K \\ Z & Y & W & T & P \end{vmatrix}$$

Plain text= HELLO

It is then made as pair.

HE		LX		LO	
H	$\rightarrow$ E	L	$\rightarrow$ Q	L	B $\rightarrow$
E	$\rightarrow$ C	X	$\rightarrow$ Z	O	X $\rightarrow$

Vigener cipher:

The key stream is the repetition of the initial secret key stream of length m. (1<=m<=26)

Example:

Plaintext- A B C D E F G H

Ks= 0, 5, 8

A	B	C	D	E	F	G	H	(B=1 =>1+5=6=>G)
0	5	8	0	5	8	0	5	
0	**6**	**10**	**3**	**9**	**13**	**6**	**12**	
A	**G**	**K**	**D**	**J**	**N**	**G**	**M**	**<=** ciphertext

Transposition cipher:

A transposition cipher is a method of encryption by which the positions held by units of plaintext (which are commonly characters or groups of characters) are shifted according to a regular system, so that the ciphertext constitutes a permutation of the plaintext. That is, the order of the units is changed.

The goal of substitution is confusion; *the transposition method is an attempt to make it difficult i.e. diffusion.*

1. Keyless transposition cipher
   There are two methods for permutation of characters
   - Text is written into a table column by column and transmitted row by row
     Example: plaintext- meet me at the park

     m e m a t e a k
     e t e t h p r

     ciphertext- memateaketethpr
   - Text is written into the table row by row and then transmitted column by column.
     Example: **m e e t**
     m e a t
     t h e p
     a r k

     ciphertext- mmtaeehreaekttp

2. Keyed transposition cipher
   Plaintext is divided into groups and permutes the character in each group.
   Example: plaintext- “enemy attack at night”

   keys:

$$\text{encryption}\ \downarrow\begin{array}{c}3\ 1\ 4\ 5\ 2\\1\ 2\ 3\ 4\ 5\end{array}\uparrow\ \text{decryption}$$

appended to make a group of 5 characters

e n e m y a t t a c k a t n i g h t y z (Group of 5 characters)
encryption: e e m y n t a a c t t k n i k t g y z h
decryption: e n e m y a t t a c k a t n i g h t y z

the characters exceeding the length of plaintext are discarded.
Like y and z two characters are discarded

3. Combining the two approaches:
   Encryption and decryption is done in three steps.
   - Text is written into a table row by row.
   - Permutation is done by reordering the column.
   - New table is read column by column

1.5 MAKING GOOD ENCRYPTION ALGORITHM

So far, the encryption algorithms we have seen are trivial, intended primarily to demonstrate the concepts of substitution and permutation. At the same time, we have examined several approaches cryptanalysts use to attack encryption algorithms. Now we examine algorithms that are widely used in the commercial world.
For each type of encryption we considered, has the advantages and disadvantages. But there is a broader question: What does it mean for a cipher to be "good"? The meaning of good depends on the intended use of the cipher. A cipher to be used by military personnel in the field has different requirements from one to be used in a secure installation with substantial computer support. In this section, we look more closely at the different characteristics of ciphers.
Shannon's Characteristics of "Good" Ciphers
In 1949, Claude Shannon [SHA49] proposed several characteristics that identify a good cipher.
1. The amount of secrecy needed should determine the amount of labor appropriate for the encryption and decryption.

2. The set of keys and the enciphering algorithm should be free from complexity.

This principle implies that we should restrict neither the choice of keys nor the types of plaintext on which the algorithm can work. For instance, an algorithm that works only on plaintext having an equal number of A's and E's is useless. Similarly, it would be difficult to select keys such that the sum of the values of the letters of the key is a prime number. Restrictions such as these make the use of the encipherment prohibitively complex. If the process is too complex, it will not be used. Furthermore, the key must be transmitted, stored, and remembered, so it must be short.

3. The implementation of the process should be as simple as possible.
Principle 3 was formulated with hand implementation in mind: A complicated algorithm is prone to error or likely to be forgotten. With the development and popularity of digital computers, algorithms far too complex for hand implementation became feasible. Still, the issue of complexity is important. People will avoid an encryption algorithm whose implementation process severely hinders message transmission, thereby undermining security. And a complex algorithm is more likely to be programmed incorrectly.

4. Errors in ciphering should not propagate and cause corruption of further information in the message.
Principle 4 acknowledges that humans make errors in their use of enciphering algorithms. One error early in the process should not throw off the entire remaining ciphertext. For example, dropping one letter in a columnar transposition throws off the entire remaining encipherment. Unless the receiver can guess where the letter was dropped, the remainder of the message will be unintelligible. By contrast, reading the wrong row or column for a polyalphabetic substitution affects only one character and remaining characters are unaffected.
5. The size of the enciphered text should be no larger than the text of the original message.

The idea behind principle 5 is that a ciphertext that expands dramatically in size cannotpossibly carry more information than the plaintext, yet it gives the cryptanalyst more data from which to infer a pattern. Furthermore, a longer ciphertext implies more space for storage and more time to communicate.

Properties of "Trustworthy" Encryption Systems
Commercial users have several requirements that must be satisfied when they select an encryption algorithm. Thus, when we say that encryption is "commercial grade," or "trustworthy," we mean that it meets these constraints:

- It is based on sound mathematics. Good cryptographic algorithms are not just invented; they are derived from solid principles.
- It has been analyzed by competent experts and found to be sound. Even the best cryptographic experts can think of only so many possible attacks, and the developers may become too convinced of the strength of their own algorithm. Thus, a review by critical outside experts is essential.
- It has stood a test of time, a as a new algorithm gains popularity, people continue to review both its mathematical foundations and the way it builds on those foundations. Although a long period of successful use and analysis is not a guarantee of a good algorithm, the flaws in many algorithms are discovered relatively soon after their release.

We can divide all the cryptography algorithms (ciphers) into two groups: symmetric key cryptography algorithms and asymmetric cryptography algorithms. Figure shows the taxonomy.

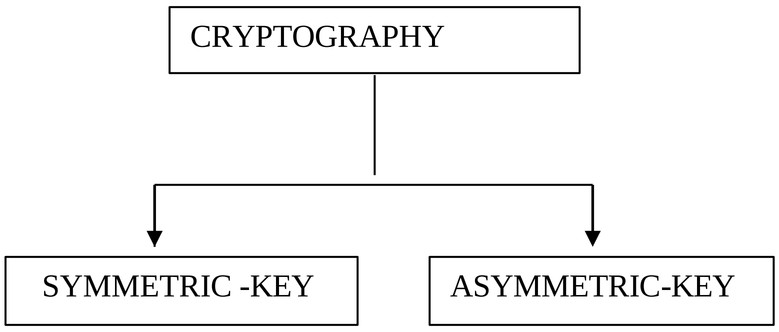


Fig :Categories of Cryptography

1. ***Symmetric Key Cryptography***

In symmetric-key cryptography, the same key is used by both parties. The sender uses this key and an encryption algorithm to encrypt data; the receiver uses the same key and the corresponding decryption algorithm to decrypt the data.

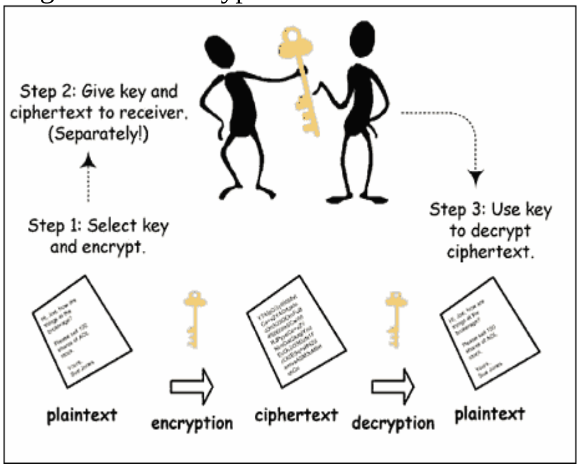

Step 2: Give key and ciphertext to receiver. (Separately!)
Step 1: Select key and encrypt.
Step 3: Use key to decrypt ciphertext.
plaintext
encryption
ciphertext
decryption
plaintext

Fig :Symmetric-key Cryptography

2. ***Asymmetric-Key Cryptography:***

In asymmetric or public-key cryptography, there are two keys: a private key and a public key. The private key is kept by the receiver. The public key is announced to the public.

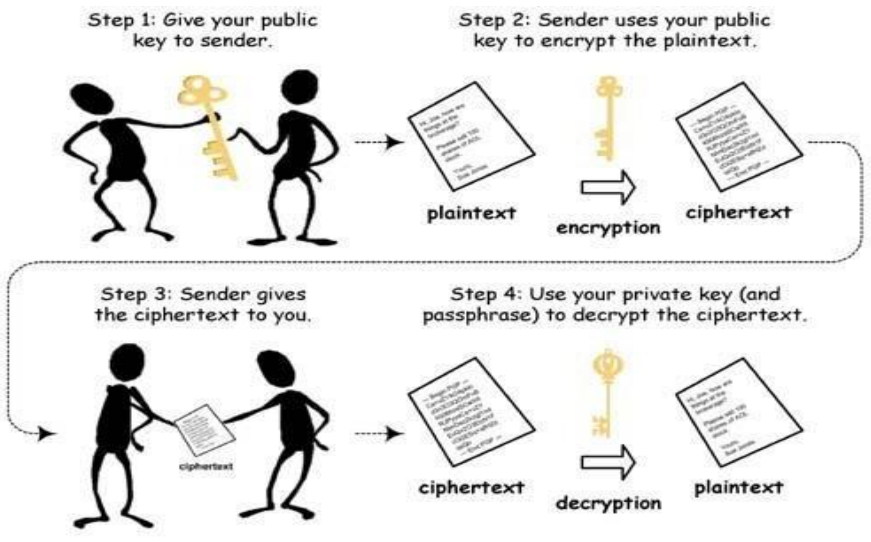

Step 1: Give your public key to sender.
Step 2: Sender uses your public key to encrypt the plaintext.
plaintext
encryption
ciphertext
Step 3: Sender gives the ciphertext to you.
Step 4: Use your private key (and passphrase) to decrypt the ciphertext.
ciphertext
ciphertext
decryption
plaintext

Fig 7:Asymmetric-key Cryptography

1.6 PRIVATE KEY CRYPTO SYSTEM

Symmetric encryption (also called *private-key encryption* or *secret-key encryption*) involves using the same key for encryption and decryption.

Encryption involves applying an operation (an algorithm) to the data to be encrypted using the private key to make them unintelligible. The slightest algorithm (such as an exclusive OR) can make the system nearly tamper proof (there being no such thing as absolute security).

However, in the 1940s, *Claude Shannon* proved that to be completely secure, private-key systems need to use keys that are at least as long as the message to be encrypted. Moreover, symmetric encryption requires that a secure channel be used to exchange the key, which seriously diminishes the usefulness of this kind of encryption system.
The main disadvantage of a secret-key cryptosystem is related to the exchange of keys. Symmetric encryption is based on the exchange of a secret (keys). The problem of key distribution therefore arises:

Moreover, a user wanting to communicate with several people while ensuring separate confidentiality levels has to use as many private keys as there are people. For a group of *N* people using a secret-key cryptosystem, it is necessary to distribute a number of keys equal to $N * (N-1) / 2$.
In the 1920s, Gilbert Vernam and Joseph Mauborgne developed the *One-Time Pad* method (sometimes called "One-Time Password" and abbreviated *OTP*), based on a randomly generated private key that is used only once and is then destroyed. During the same period, the Kremlin and the White House were connected by the famous **red telephone**, that is, a

telephone where calls were encrypted thanks to a private key according to the *one-time pad* method. The private key was exchanged thanks to the diplomatic bag (playing the role of secure channel).

An important distinction in symmetric cryptographic algorithms is between stream and block ciphers.

Stream cipher: Stream ciphers convert one symbol of plaintext directly into a symbol of ciphertext.

Advantages:

- Speed of transformation: algorithms are linear in time and constant in space.
- Low error propagation: an error in encrypting one symbol likely will not affect subsequent symbols.

Disadvantages:

- Low diffusion: all information of a plaintext symbol is contained in a single ciphertext symbol.
- Susceptibility to insertions/ modifications: an active interceptor who breaks the algorithm might insert spurious text that looks authentic.

Block ciphers: It encrypt a group of plaintext symbols as one block.

Advantages:

- High diffusion: information from one plaintext symbol is diffused into several ciphertext symbols.
- Immunity to tampering: difficult to insert symbols without detection.

Disadvantages:

- Slowness of encryption: an entire block must be accumulated before encryption / decryption can begin.
- Error propagation: An error in one symbol may corrupt the entire block.

Simple substitution is an example of a stream cipher. Columnar transposition is a block cipher.

1.7 THE DATA ENCRYPTION STANDARD

The Data Encryption Standard (DES), a system developed for the U.S. government, was intended for use by the general public. It has been officially accepted as a cryptographic standard both in the United States and abroad.

The DES algorithm is a careful and complex combination of two fundamental building blocks of encryption: substitution and transposition. The algorithm derives its strength from repeated application of these two techniques, one on top of the other, for a total of 16 cycles. The sheer complexity of tracing a single bit through 16 iterations of substitutions and transpositions has so far stopped researchers in the public from identifying more than a handful of general properties of the algorithm. The algorithm begins by encrypting the plaintext as blocks of 64 bits. The key is 64 bits long, but in fact it can be any 56-bit number. (The extra 8 bits are often used as check digits and do not affect encryption in normal implementations.) The user can change the key at will any time there is uncertainty about the

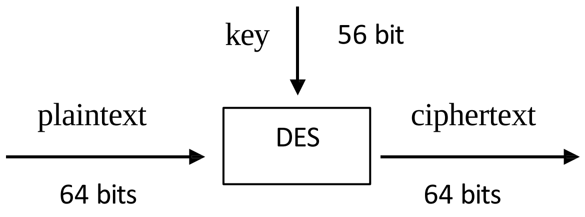
Features: –

Block size = 64 bits

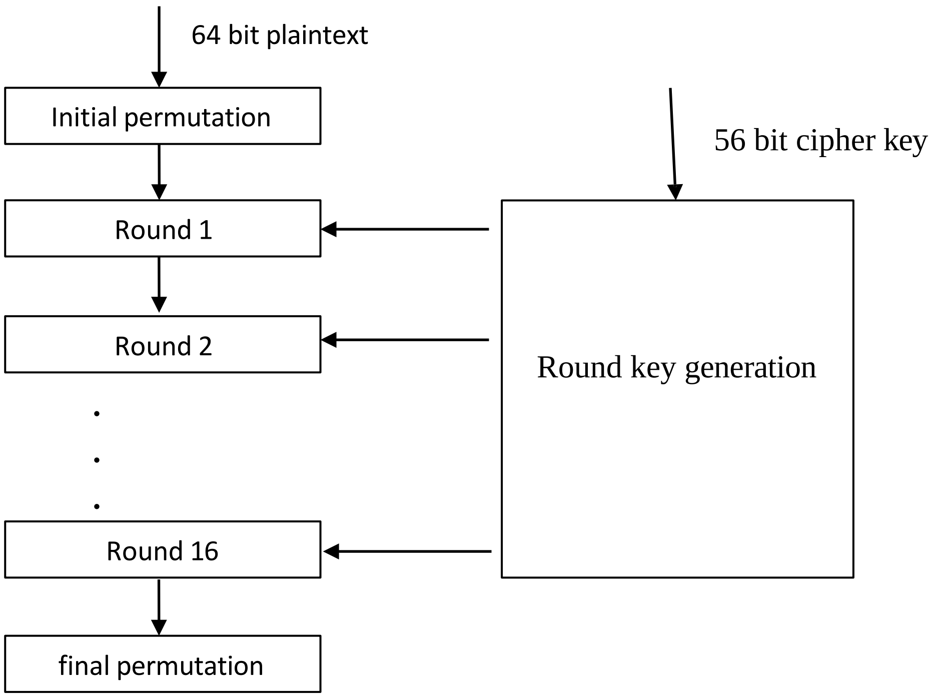
– Key size = 56 bits (in reality, 64 bits, but 8 are used as parity-check bits for error control, see next slide)

– Number of rounds = 16

– 16 intermediary keys, each 48 bits



Working principle:



The Feistel (F) function

The F-function, depicted in Figure 2, operates on half a block (32 bits) at a time and consists of four stages:

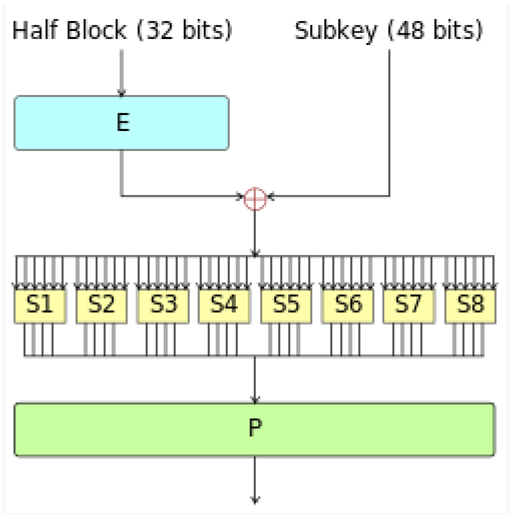


Fig: —The Feistel function (F-function) of DES

1. *Expansion* — the 32-bit half-block is expanded to 48 bits using the *expansion permutation*, denoted *E* in the diagram, by duplicating half of the bits. The output consists of eight 6-bit (8 * 6 = 48 bits) pieces, each containing a copy of 4 corresponding input bits, plus a copy of the immediately adjacent bit from each of the input pieces to either side.
2. *Key mixing* — the result is combined with a *subkey* using an XOR operation. 16 48- bit subkeys — one for each round — are derived from the main key using the *key schedule* (described below).
3. *Substitution* — after mixing in the subkey, the block is divided into eight 6-bit pieces before processing by the *S-boxes*, or *substitution boxes*. Each of the eight S-boxes replaces its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table. The S-boxes provide the core of the security of DES — without them, the cipher would be linear, and trivially breakable.
4. *Permutation* — finally, the 32 outputs from the S-boxes are rearranged according to a fixed permutation, the *P-box*. This is designed so that, after permutation, each S-box's output bits are spread across 4 different S boxes in the next round.

The alternation of substitution from the S-boxes, and permutation of bits from the P-box and E-expansion provides so-called "confusion and diffusion" respectively, a concept identified by Claude Shannon in the 1940s as a necessary condition for a secure yet practical cipher.

Key schedule

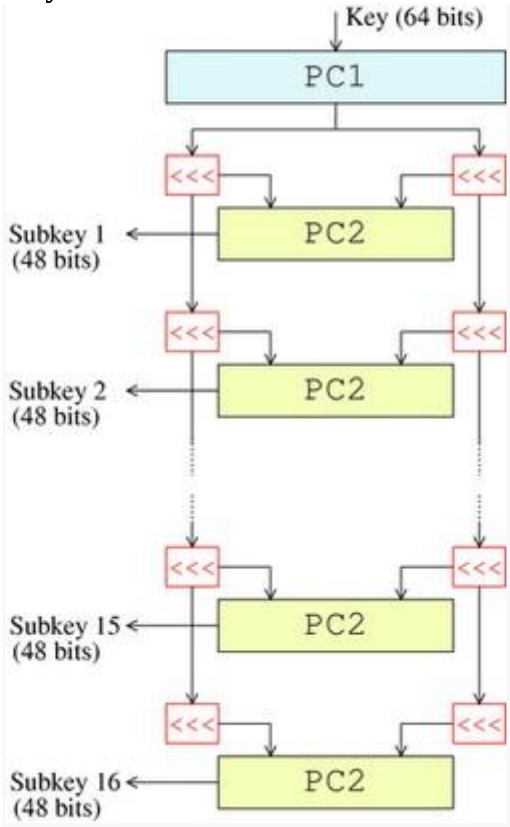


Fig:— The key-schedule of DES

Figure illustrates the *key schedule* for encryption — the algorithm which generates the subkeys. Initially, 56 bits of the key are selected from the initial 64 by *Permuted Choice 1* (*PC-1*) — the remaining eight bits are either discarded or used as parity check bits. The 56 bits are then divided into two 28-bit halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left by one or two bits (specified for each round), and then 48 subkey bits are selected by *Permuted Choice 2* (*PC-2*) — 24 bits from the left half, and 24 from the right. The rotations (denoted by "<<<" in the diagram) mean that a different set of bits is used in each subkey; each bit is used in approximately 14 out of the 16 subkeys.

The key schedule for decryption is similar — the subkeys are in reverse order compared to encryption. Apart from that change, the process is the same as for encryption. The same 28 bits are passed to all rotation boxes.

Security of the DES

Since it was first announced, DES has been controversial. Many researchers have questioned the security it provides. Much of this controversy has appeared in the open literature, but certain DES features have neither been revealed by the designers nor inferred by outside analysts.
In 1990, Biham and Shamir invented a technique, differential cryptanalysis, that investigates the change in algorithmic strength when an encryption algorithm is changed in some way. In 1991 they applied their technique to DES, showing that almost any change to the algorithm weakens it. Their changes included cutting the number of iterations from 16 to 15, changing the expansion or substitution rule, or altering the order of an iteration. In each

case, when they weakened the algorithm, Biham and Shamir could break the modified version. Thus, it seems as if the design of DES is optimal.
However, Diffie and Hellman argued in 1977 that a 56-bit key is too short. In 1977, it was prohibitive to test all 256 (approximately 1015) keys on then current computers. But they argued that over time, computers would become more powerful and the DES algorithm would remain unchanged; eventually, the speed of computers would exceed the strength of DES. Exactly that has happened. In 1997 researchers using over 3,500 machines in parallel were able to infer a DES key in four months' work. And in 1998 for approximately $100,000, researchers built a special "DES cracker" machine that could find a DES key in approximately four days.
In 1995, the U.S. National Institute of Standards and Technology (NIST, the renamed NBS) began the search for a new, strong encryption algorithm. The response to that search has become the Advanced Encryption Standard, or AES.

1.8 The AES Encryption Algorithm

The AES is likely to be the commercial-grade symmetric algorithm of choice for years, if not decades. Let us look at it more closely.
The AES Contest
In January 1997, NIST called for cryptographers to develop a new encryption system. As with the call for candidates from which DES was selected, NIST made several important restrictions. The algorithms had to be

- Unclassified
- publicly disclosed
- available royalty-free for use worldwide
- symmetric block cipher algorithms, for blocks of 128 bits
- usable with key sizes of 128, 192, and 256 bits

AES is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation, and is fast in both software and hardware.[9]Unlike its predecessor DES, AES does not use a Feistel network. AES is a variant of Rijndael which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits. By contrast, the Rijndael specification *per se* is specified with block and key sizes that may be any multiple of 32 bits, both with a minimum of 128 and a maximum of 256 bits.

AES operates on a 4×4 column-major order matrix of bytes, termed the *state*, although some versions of Rijndael have a larger block size and have additional columns in the state. Most AES calculations are done in a special finite field.

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The number of cycles of repetition are as follows:

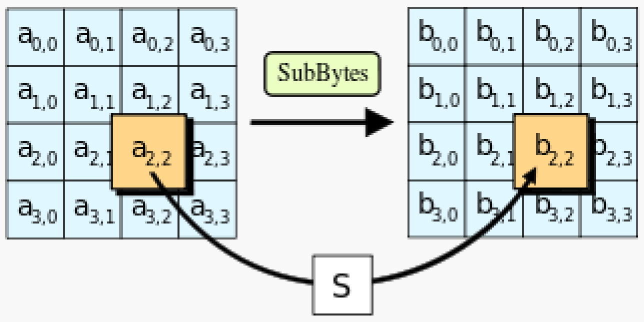
- 10 cycles of repetition for 128-bit keys.
- 12 cycles of repetition for 192-bit keys.
- 14 cycles of repetition for 256-bit keys.

Each round consists of several processing steps, each containing four similar but different stages, including one that depends on the encryption key itself. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

High-level description of the algorithm

1. Key Expansions—round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.
2. Initial Round
    1. AddRoundKey—each byte of the state is combined with a block of the round key using bitwise xor.
3. Rounds
    1. SubBytes—a non-linear substitution step where each byte is replaced with another according to a lookup table.
    2. ShiftRows—a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
    3. MixColumns—a mixing operation which operates on the columns of the state, combining the four bytes in each column.
    4. AddRoundKey
4. Final Round (no MixColumns)
    1. SubBytes
    2. ShiftRows
    3. AddRoundKey.

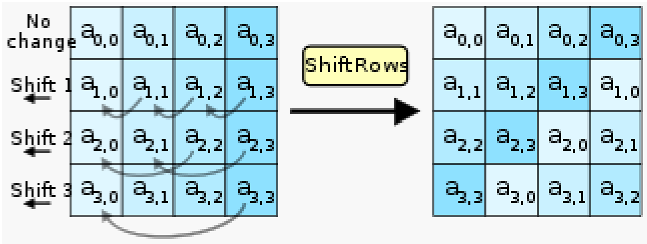
The SubBytes steps


$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
SubBytes
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
S

In the SubBytes step, each byte in the state is replaced with its entry in a fixed 8-bit lookup table, *S*; $b_{ij} = S(a_{ij})$.

In the SubBytes step, each byte $a_{i,j}$ in the *state* matrix is replaced with a SubByte $S(a_{i,j})$ using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher. The S-box used is derived from the multiplicative inverse over **GF**(2^8), known to have good non-linearity properties. To avoid attacks based on simple algebraic properties, the S-box is constructed by combining the inverse function with an invertible affine transformation. The S-box is also chosen to avoid any fixed points (and so is a derangement), i.e., $S(a_{i,j}) \neq a_{i,j}$, and also any opposite fixed points, i.e., $S(a_{i,j}) \oplus a_{i,j} \neq 0\text{xFF}$. While performing the decryption, Inverse SubBytes step is used, which requires first taking the affine transformation and then finding the multiplicative inverse (just reversing the steps used in SubBytes step).

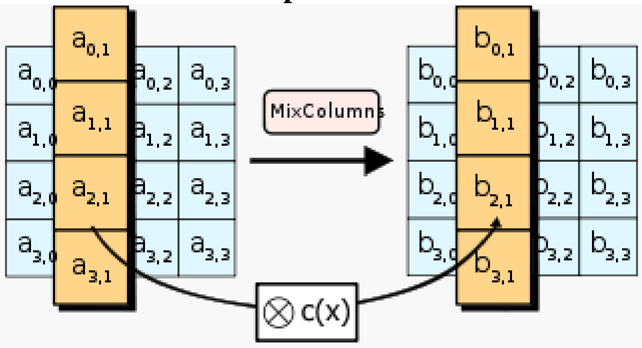
The ShiftRows step



In the ShiftRows step, bytes in each row of the state are shifted cyclically to the left. The number of places each byte is shifted differs for each row.

The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. For blocks of sizes 128 bits and 192 bits, the shifting pattern is the same. Row n is shifted left circular by n-1 bytes. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state. (Rijndael variants with a larger block size have slightly different offsets). For a 256-bit block, the first row is unchanged and the shifting for the second, third and fourth row is 1 byte, 3 bytes and 4 bytes respectively—this change only applies for the Rijndael cipher when used with a 256-bit block, as AES does not use 256-bit blocks. The importance of this step is to avoid the columns being linearly independent, in which case, AES degenerates into four independent block ciphers.

The MixColumns step



In the MixColumns step, each column of the state is multiplied with a fixed polynomial *c(x)*.

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. Together with ShiftRows, MixColumns provides diffusion in the cipher.

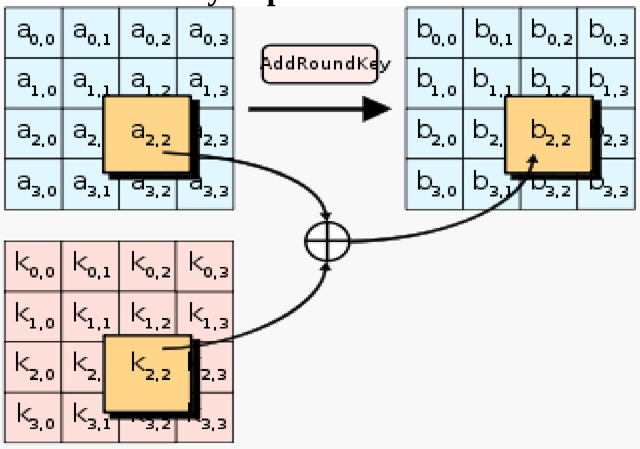
During this operation, each column is multiplied by a fixed matrix:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Matrix multiplication is composed of multiplication and addition of the entries, and here the multiplication operation can be defined as this: multiplication by 1 means no change, multiplication by 2 means shifting to the left, and multiplication by 3 means shifting to the left and then performing XOR with the initial unshifted value. After shifting, a conditional XOR with 0x1B should be performed if the shifted value is larger than 0xFF. (These are special cases of the usual multiplication in **GF**. Addition is simply XOR.

In more general sense, each column is treated as a polynomial over **GF**and is then multiplied modulo x^4+1 with a fixed polynomial $c(x) = 0x03 \cdot x^3 + x^2 + x + 0x02$. The coefficients are displayed in their hexadecimal equivalent of the binary representation of bit polynomials from **GF**(2)[x]. The MixColumns step can also be viewed as a multiplication by the shown particular MDS matrix in the finite field **GF**(2^8). This process is described further in the article Rijndael mix columns.

The AddRoundKey step



In the AddRoundKey step, each byte of the state is combined with a byte of the round subkey using the XORoperation (⊕).

In the AddRoundKey step, the subkey is combined with the state. For each round, a subkey is derived from the main keyusing Rijndael's key schedule; each subkey is the same size as the state. The subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.

Optimization of the cipher

On systems with 32-bit or larger words, it is possible to speed up execution of this cipher by combining the SubBytes andShiftRows steps with the MixColumns step by transforming them into a sequence of table lookups. This requires four 256-entry 32-bit tables, and utilizes a total of four kilobytes (4096 bytes) of memory — one kilobyte for each table. A round can then be done with 16 table lookups and 12 32-bit exclusive-or operations, followed by four 32-bit exclusive-or operations in the AddRoundKey steps.

If the resulting four-kilobyte table size is too large for a given target platform, the table lookup operation can be performed with a single 256-entry 32-bit (i.e. 1 kilobyte) table by the use of circular rotates.

Using a byte-oriented approach, it is possible to combine the SubBytes, ShiftRows, and MixColumns steps into a single round operation.

1.9 PUBLIC KEY CRYPTOSYSTEM

Public-key cryptography, also known as **asymmetric cryptography**, is a class of cryptographic algorithms which requires two separate keys, one of which is *secret* (or *private*) and one of which is *public*.

Public-key cryptography is often used to secure electronic communication over an open networked environment such as the internet. Open networked environments are susceptible to a variety of communication security problems such as man-in-the-middle attacks and other security threats. Sending a secure communication means that the communication being sent must not be readable during transit (preserving confidentiality), the communication must not be modified during transit (preserving the integrity of the communication) and to enforce non-repudiation or non-denial of the sending of the communication. Combining public-key cryptography with an Enveloped Public Key Encryption (EPKE) method, allows for the secure sending of a communication over an open networked environment.

The distinguishing technique used in public-key cryptography is the use of asymmetric key algorithms, where the key used to encrypt a message is not the same as the key used to decrypt it. Each user has a pair of cryptographic keys – a **public encryption key** and a **private decryption key**. Similarly, a key pair used for digital signatures consists of a**private signing key** and a **public verification key**. The public key is widely distributed, while the private key is known only to its proprietor. The keys are related mathematically, but the parameters are chosen so that calculating the private key from the public key is either impossible or prohibitively expensive.

In contrast, symmetric-key algorithms – variations of which have been used for thousands of years – use a *single* secret key, which must be shared and kept private by both the sender and the receiver, for both encryption and decryption. To use a symmetric encryption scheme, the sender and receiver must securely share a key in advance.

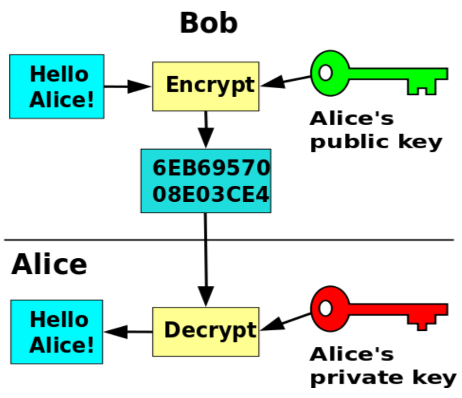


Fig : Public Key cryptosystem

1.10 USE OF ENCRYPTION

Encryption has long been used by militaries and governments to facilitate secret communication. It is now commonly used in protecting information within many kinds of civilian systems. For example, the Computer Security Institute reported that in 2007, 71% of companies surveyed utilized encryption for some of their data in transit, and 53% utilized encryption for some of their data in storage.Encryption can be used to protect data "at rest", such as files on computers and storage devices (e.g. USB flash drives). In recent years there have been numerous reports of confidential data such as customers' personal records being exposed through loss or theft of laptops or backup drives. Encrypting such files at rest helps protect them should physical security measures fail. Digital rights management systems, which prevent unauthorized use or reproduction of copyrighted material and protect software against reverse engineering (see also copy protection), is another somewhat different example of using encryption on data at rest.

Encryption is also used to protect data in transit, for example data being transferred via networks (e.g. the Internet, e-commerce), mobile telephones, wireless microphones,wireless intercom systems, Bluetooth devices and bank automatic teller machines. There have been numerous reports of data in transit being intercepted in recent years.Encrypting data in transit also helps to secure it as it is often difficult to physically secure all access to networks.

Let us look more closely at four applications of encryption: cryptographic hash functions, key exchange, digital signatures, and certificates.

Message verification

Encryption, by itself, can protect the confidentiality of messages, but other techniques are still needed to protect the integrity and authenticity of a message; for example, verification of a message authentication code (MAC) or a digital signature. Standards for cryptographic software and hardware to perform encryption are widely available, but successfully using encryption to ensure security may be a challenging problem. A single error in system design or execution can allow successful attacks. Sometimes an adversary can obtain unencrypted information without directly undoing the encryption. See, e.g., traffic analysis, TEMPEST, or Trojan horse.

Digital signature and encryption must be applied to the ciphertext when it is created (typically on the same device used to compose the message) to avoid tampering; otherwise any node between the sender and the encryption agent could potentially tamper with it. Encrypting at the time of creation is only secure if the encryption device itself has not been tampered with.

PSEUDO-RANDOMNESS

Forcryptography, the use of pseudorandom number generators (whether hardware or software or some combination) is insecure. When random values are required in cryptography, the goal is to make a message as hard to crack as possible, by eliminating or obscuring the parameters used to encrypt the message (the key) from the message itself or from the context in which it is carried. Pseudorandom sequences are deterministic and reproducible; all that is required in order to discover and reproduce a pseudorandom sequence is the algorithm used to generate it and the initial seed. So the entire sequence of numbers is only as powerful as the randomly chosen parts - sometimes the algorithm and the seed, but usually only the seed.

There are many examples in cryptographic history of cyphers, otherwise excellent, in which random choices were not random enough and security was lost as a direct consequence. The World War II Japanese PURPLE cypher machine used for diplomatic communications is a good example. It was consistently broken throughout WWII, mostly because the "key values" used were insufficiently random. They had patterns, and those patterns made any intercepted traffic readily decryptable. Had the keys (i.e. the initial settings of the stepping switches in the machine) been made unpredictably (i.e. randomly), that traffic would have been much harder to break, and perhaps even secure in practice.

Users and designers of cryptography are strongly cautioned to treat their randomness needs with the utmost care. Absolutely nothing has changed with the era of computerized cryptography, except that patterns in pseudorandom data are easier to discover than ever before. Randomness is, if anything, more important than ever.

HASHING

A **cryptographic hash function** is a hash function which is considered practically impossible to invert, that is, to recreate the input data from its hash value alone. These one-way hash functions have been called "the workhorses of modern cryptography". The input data is often called the *message*, and the hash value is often called the *message digest* or simply the *digest*.

The ideal cryptographic hash function has four main properties:

- it is easy to compute the hash value for any given message
- it is infeasible to generate a message that has a given hash
- it is infeasible to modify a message without changing the hash
- it is infeasible to find two different messages with the same hash.

Cryptographic hash functions have many information security applications, notably in digital signatures, message authentication codes (MACs), and other forms of authentication. They can also be used as ordinary hash functions, to index data in hash tables, for fingerprinting, to detect duplicate data or uniquely identify files, and as check sums to detect accidental data corruption. Indeed, in information security contexts, cryptographic hash values are sometimes called (*digital*) *fingerprints*, *checksums*, or just *hash values*, even though all these terms stand for more general functions with rather different properties and purposes.

Most cryptographic hash functions are designed to take a string of any length as input and produce a fixed-length hash value.

A cryptographic hash function must be able to withstand all known types of cryptanalytic attack. At a minimum, it must have the following properties:

- *Pre-image resistance*

  Given a hash h it should be difficult to find any message m such that $h = hash(m)$. This concept is related to that of one-way function. Functions that lack this property are vulnerable to preimage attacks.

- *Second pre-image resistance*

  Given an input m_1 it should be difficult to find another input m_2 such that $m_1 \neq m_2$ and $hash(m_1) = hash(m_2)$. Functions that lack this property are vulnerable to second-preimage attacks.

- *Collision resistance*

  It should be difficult to find two different messages m_1 and m_2 such that $hash(m_1) = hash(m_2)$. Such a pair is called a cryptographic hash collision. This property is sometimes referred to as *strong collision resistance.* It requires a hash value at least twice as long as that required for preimage-resistance; otherwise collisions may be found by a birthday attack.

These properties imply that a malicious adversary cannot replace or modify the input data without changing its digest. Thus, if two strings have the same digest, one can be very confident that they are identical.

A function meeting these criteria may still have undesirable properties. Currently popular cryptographic hash functions are vulnerable to *length-extension* attacks: given $hash(m)$and $len(m)$ but not m, by choosing a suitable m' an attacker can calculate $hash(m \| m')$ where $\|$ denotes concatenation.[2] This property can be used to break naive authentication schemes based on hash functions. The HMAC construction works around these problems.

Ideally, one may wish for even stronger conditions. It should be impossible for an adversary to find two messages with substantially similar digests; or to infer any useful information about the data, given only its digest. Therefore, a cryptographic hash function should behave as much as possible like a random function while still being deterministic and efficiently computable.

Checksum algorithms, such as CRC32 and other cyclic redundancy checks, are designed to meet much weaker requirements, and are generally unsuitable as cryptographic hash functions. For example, a CRC was used for message integrity in the WEP encryption standard, but an attack was readily discovered which exploited the linearity of the checksum.

MODULE 2

PROGRAM SECURITY

2.1 SECURE PROGRAM

Consider what we mean when we say that a program is "secure." We know that security implies some degree of trust that the program enforces expected confidentiality, integrity, and availability. From the point of view of a program or a programmer, how can we look at a software component or code fragment and assess its security? This question is, of course, similar to the problem of assessing software quality in general. One way to assess security or quality is to ask people to name the characteristics of software that contribute to its overall security. However, we are likely to get different answers from different people. This difference occurs because the importance of the characteristics depends on who is analyzing the software. For example, one person may decide that code is secure because it takes too long to break through its security controls. And someone else may decide code is secure if it has run for a period of time with no apparent failures. But a third person may decide that any potential fault in meeting security requirements makes code insecure.

Early work in computer security was based on the paradigm of "penetrate and patch," in which analysts searched for and repaired faults. Often, a top-quality "tiger team" would be convened to test a system's security by attempting to cause it to fail. The test was considered to be a "proof" of security; if the system withstood the attacks, it was considered secure. Unfortunately, far too often the proof became a counterexample, in which not just one but several serious security problems were uncovered. The problem discovery in turn led to a rapid effort to "patch" the system to repair or restore the security. However, the patch efforts were largely useless, making the system less secure rather than more secure because they frequently introduced new faults. There are at least four reasons why.

1. The pressure to repair a specific problem encouraged a narrow focus on the fault itself and not on its context. In particular, the analysts paid attention to the immediate cause of the failure and not to the underlying design or requirements faults.

2. The fault often had nonobvious side effects in places other than the immediate area of the fault.

3. Fixing one problem often caused a failure somewhere else, or the patch addressed the problem in only one place, not in other related places.

4. The fault could not be fixed properly because system functionality or performance would suffer as a consequence.

The inadequacies of penetrate-and-patch led researchers to seek a better way to be confident that code meets its security requirements. One way to do that is to compare the requirements with the behavior. That is, to understand program security, we can examine programs to see whether they behave as their designers intended or users expected. We call such unexpected behavior a <u>program security flaw</u>; it is inappropriate program behaviour caused by a program vulnerability.
Program security flaws can derive from any kind of software fault. That is, they cover everything from a misunderstanding of program requirements to a one-character error in coding or even typing. The flaws can result from problems in a single code component or from the failure of several programs or program pieces to interact compatibly through a shared interface. The security flaws can reflect code that was intentionally designed or coded

to be malicious or code that was simply developed in a sloppy or misguided way. Thus, it makes sense to divide program flaws into two separate logical categories: inadvertent human errors versus malicious, intentionally induced flaws.

Types of Flaws

To aid our understanding of the problems and their prevention or correction, we can define categories that distinguish one kind of problem from another. For example, Landwehr et al. present a taxonomy of program flaws, dividing them first into intentional and inadvertent flaws. They further divide intentional flaws into malicious and no malicious ones.

In the taxonomy, the inadvertent flaws fall into six categories:

- validation error (incomplete or inconsistent): permission checks
- domain error: controlled access to data
- serialization and aliasing: program flow order
- inadequate identification and authentication: basis for authorization
- boundary condition violation: failure on first or last case
- other exploitable logic errors

2.2. NON MALICIOUS PROGRAM ERRORS

Being human, programmers and other developers make many mistakes, most of which are unintentional and nonmalicious. Many such errors cause program malfunctions but do not lead to more serious security vulnerabilities. However, a few classes of errors have plagued programmers and security professionals for decades, and there is no reason to believe they will disappear. In this section we consider three classic error types that have enabled many recent security breaches. We explain each type, why it is relevant to security, and how it can be prevented or mitigated.

Buffer Overflows

A buffer overflow is the computing equivalent of trying to pour two liters of water into a one-liter pitcher: Some water is going to spill out and make a mess. And in computing, what a mess these errors have made!

Definition

A buffer (or array or string) is a space in which data can be held. A buffer resides in memory. Because memory is finite, a buffer's capacity is finite. For this reason, in many programming languages the programmer must declare the buffer's maximum size so that the compiler can set aside that amount of space.

Let us look at an example to see how buffer overflows can happen. Suppose a C language program contains the declaration:

char sample[10];

The compiler sets aside 10 bytes to store this buffer, one byte for each of the ten elements of the array, sample[0] through sample[9]. Now we execute the statement:

sample[10] = 'A';

The subscript is out of bounds (that is, it does not fall between 0 and 9), so we have a problem.The nicest outcome (from a security perspective) is for the compiler to detect the problem and mark the error during compilation. However, if the statement were

sample[i] = 'A';

we could not identify the problem until i was set during execution to a too-big subscript. It would be useful if, during execution, the system produced an error message warning of a subscript out of bounds. Unfortunately, in some languages, buffer sizes do not have to be predefined, so there is no way to detect an out-of-bounds error. More importantly, the code needed to check each subscript against its potential maximum value takes time and space during execution, and the resources are applied to catch a problem that occurs relatively infrequently. Even if the compiler were careful in analyzing the buffer declaration and use, this same problem can be caused with pointers, for which there is no reasonable way to define a proper limit. Thus, some compilers do not generate the code to check for exceeding bounds. Let us examine this problem more closely. It is important to recognize that the potential overflow causes a serious problem only in some instances. The problem's occurrence depends on what is adjacent to the array sample. For example, suppose each of the ten elements of the array sample is filled with the letter A and the erroneous reference uses the letter B, as follows:

for (i=0; i<=9; i++) sample[i] = 'A'; sample[10] = 'B'

All program and data elements are in memory during execution, sharing space with the operating system, other code, and resident routines. So there are four cases to consider in deciding where the 'B' goes. If the extra character overflows into the user's data space, it simply overwrites an existing variable value (or it may be written into an as-yet unused location), perhaps affecting the program's result, but affecting no other program or data.

3.3 VIRUS AND OTHER MALICIOUS CODE

By themselves, programs are seldom security threats. The programs operate on data, taking action only when data and state changes trigger it. Much of the work done by a program is invisible to users, so they are not likely to be aware of any malicious activity. For instance, when was the last time you saw a bit? Do you know in what form a document file is stored? If you know a document resides somewhere on a disk, can you find it? Can you tell if a game program does anything in addition to its expected interaction with you? Which files are modified by a word processor when you create a document? Most users cannot answer these questions. However, since computer data are not usually seen directly by users, malicious people can make programs serve as vehicles to access and change data and other programs. Let us look at the possible effects of malicious code and then examine in detail several kinds of programs that can be used for interception or modification of data.

Why Worry About Malicious Code?

None of us likes the unexpected, especially in our programs. Malicious code behaves in unexpected ways, thanks to a malicious programmer's intention. We think of the malicious code as lurking inside our system: all or some of a program that we are running or even a nasty part of a separate program that somehow attaches itself to another (good) program.

Malicious Code Can Do Much (Harm)
Malicious code can do anything any other program can, such as writing a message on a computer screen, stopping a running program, generating a sound, or erasing a stored file. Or malicious code can do nothing at all right now; it can be planted to lie dormant, undetected, until some event triggers the code to act. The trigger can be a time or date, an interval (for example, after 30 minutes), an event (for example, when a particular program is executed), a condition (for example, when communication occurs on a modem), a count (for example, the fifth time something happens), some combination of these, or a random situation. In fact, malicious code can do different things each time, or nothing most of the time with something dramatic on occasion. In general, malicious code can act with all the predictability of a two-year-old child: We know in general what two-year-olds do, we may even know what a specific two-year-old often does in certain situations, but two-year-olds have an amazing capacity to do the unexpected.

Malicious code runs under the user's authority. Thus, malicious code can touch everything the user can touch, and in the same ways. Users typically have complete control over their own program code and data files; they can read, write, modify, append, and even delete them. And well they should. But malicious code can do the same, without the user's permission or even knowledge.

Malicious Code Has Been Around a Long Time
The popular literature and press continue to highlight the effects of malicious code as if it were a relatively recent phenomenon. It is not. Cohen [COH84] is sometimes credited with the discovery of viruses, but in fact Cohen gave a name to a phenomenon known long before. For example, Thompson, in his 1984 Turing Award lecture, "Reflections on Trusting Trust" [THO84], described code that can be passed by a compiler. In that lecture, he refers to an earlier Air Force document, the Multics security evaluation [KAR74, KAR02]. In fact, references to virus behavior go back at least to 1970. Ware's 1970 study (publicly released in 1979 [WAR79]) and Anderson's planning study for the U.S. Air Force [AND72] (to which Schell also refers) *still* accurately describe threats, vulnerabilities, and program security flaws, especially intentional ones. What *is* new about malicious code is the number of distinct instances and copies that have appeared.

So malicious code is still around, and its effects are more pervasive. It is important for us to learn what it looks like and how it works, so that we can take steps to prevent it from doing damage or at least mediate its effects. How can malicious code take control of a system? How can it lodge in a system? How does malicious code spread? How can it be recognized? How can it be detected? How can it be stopped? How can it be prevented? We address these questions in the following sections.

Kinds of Malicious Code
Malicious code or a **rogue program** is the general name for unanticipated or undesired effects in programs or program parts, caused by an agent intent on damage. This definition eliminates unintentional errors, although they can also have a serious negative effect. This definition also excludes coincidence, in which two benign programs combine for a negative

effect. The **agent** is the writer of the program or the person who causes its distribution. By this definition, most faults found in software inspections, reviews, and testing do not qualify as malicious code, because we think of them as unintentional. However, keep in mind as you read this chapter that unintentional faults can in fact invoke the same responses as intentional malevolence; a benign cause can still lead to a disastrous effect.

You are likely to have been affected by a virus at one time or another, either because your computer was infected by one or because you could not access an infected system while its administrators were cleaning up the mess one made. In fact, your virus might actually have been a worm: The terminology of malicious code is sometimes used imprecisely. A **virus** is a program that can pass on malicious code to other nonmalicious programs by modifying them. The term "virus" was coined because the affected program acts like a biological virus: It infects other healthy subjects by attaching itself to the program and either destroying it or coexisting with it. Because viruses are insidious, we cannot assume that a clean program yesterday is still clean today. Moreover, a good program can be modified to include a copy of the virus program, so the infected good program itself begins to act as a virus, infecting other programs. The infection usually spreads at a geometric rate, eventually overtaking an entire computing system and spreading to all other connected systems.

A virus can be either transient or resident. A **transient** virus has a life that depends on the life of its host; the virus runs when its attached program executes and terminates when its attached program ends. (During its execution, the transient virus may have spread its infection to other programs.) A**resident** virus locates itself in memory; then it can remain active or be activated as a stand-alone program, even after its attached program ends.

A **Trojan horse** is malicious code that, in addition to its primary effect, has a second, nonobvious malicious effect.[1] As an example of a computer Trojan horse,

A **logic bomb** is a class of malicious code that "detonates" or goes off when a specified condition occurs. A **time bomb** is a logic bomb whose trigger is a time or date.

A **trapdoor** or **backdoor** is a feature in a program by which someone can access the program other than by the obvious, direct call, perhaps with special privileges. For instance, an automated bank teller program might allow anyone entering the number 990099 on the keypad to process the log of everyone's transactions at that machine. In this example, the trapdoor could be intentional, for maintenance purposes, or it could be an illicit way for the implementer to wipe out any record of a crime.

A **worm** is a program that spreads copies of itself through a network. The primary difference between a worm and a virus is that a worm operates through networks, and a virus can spread through any medium (but usually uses copied program or data files). Additionally, the worm spreads copies of itself as a stand-alone program, whereas the virus spreads copies of itself as a program that attaches to or embeds in other programs.

White et al. also define a **rabbit** as a virus or worm that self-replicates without bound, with the intention of exhausting some computing resource. A rabbit might create copies of itself and store them on disk, in an effort to completely fill the disk, for example.

These definitions match current careful usage. The distinctions among these terms are small, and often the terms are confused, especially in the popular press. The term "virus" is often used to refer to any piece of malicious code. Furthermore, two or more forms of malicious code can be combined to produce a third kind of problem. For instance, a virus can be a time bomb if the viral code that is spreading will trigger an event after a period of time has passed. The kinds of malicious code are summarized in Table 3-1.

TABLE 3-1 Types of Malicious Code.

Code Type	**Characteristics**
Virus	Attaches itself to program and propagates copies of itself to other programs
Trojan horse	Contains unexpected, additional functionality
Logic bomb	Triggers action when condition occurs
Time bomb	Triggers action when specified time occurs
Trapdoor	Allows unauthorized access to functionality
Worm	Propagates copies of itself through a network
Rabbit	Replicates itself without limit to exhaust resource

Because "virus" is the popular name given to all forms of malicious code and because fuzzy lines exist between different kinds of malicious code, we will not be too restrictive in the following discussion. We want to look at how malicious code spreads, how it is activated, and what effect it can have. A virus is a convenient term for mobile malicious code, and so in the following sections we use the term "virus" almost exclusively. The points made apply also to other forms of malicious code.

How Viruses Attach

A printed copy of a virus does nothing and threatens no one. Even executable virus code sitting on a disk does nothing. What triggers a virus to start replicating? For a virus to do its malicious work and spread itself, it must be activated by being executed. Fortunately for virus writers, but unfortunately for the rest of us, there are many ways to ensure that programs will be executed on a running computer.

For example, recall the SETUP program that you initiate on your computer. It may call dozens or hundreds of other programs, some on the distribution medium, some already residing on the computer, some in memory. If any one of these programs contains a virus, the virus code could be activated. Let us see how. Suppose the virus code were in a program on the distribution medium, such as a CD; when executed, the virus could install itself on a permanent storage medium (typically, a hard disk), and also in any and all executing programs in memory. Human intervention is necessary to start the process; a human being puts the virus on the distribution medium, and perhaps another initiates the execution of the program to which the virus is attached. (It is possible for execution to occur without human intervention, though, such as when execution is triggered by a date or the passage of a certain amount of time.) After that, no human intervention is needed; the virus can spread by itself.

A more common means of virus activation is as an attachment to an e-mail message. In this attack, the virus writer tries to convince the victim (the recipient of an e-mail message) to open the attachment. Once the viral attachment is opened, the activated virus can do its work. Some modern e-mail handlers, in a drive to "help" the receiver (victim), will automatically open attachments as soon as the receiver opens the body of the e-mail message. The virus can be executable code embedded in an executable attachment, but other types of files are equally dangerous. For example, objects such as graphics or photo images can contain code to be executed by an editor, so they can be transmission agents for viruses. In general, it is safer to force users to open files on their own rather than automatically; it is a bad idea for programs to perform potentially security-relevant actions without a user's consent.

Appended Viruses
A program virus attaches itself to a program; then, whenever the program is run, the virus is activated. This kind of attachment is usually easy to program.

In the simplest case, a virus inserts a copy of itself into the executable program file before the first executable instruction. Then, all the virus instructions execute first; after the last virus instruction, control flows naturally to what used to be the first program instruction.

Virus Appended to a Program.

This kind of attachment is simple and usually effective. The virus writer does not need to know anything about the program to which the virus will attach, and often the attached program simply serves as a carrier for the virus. The virus performs its task and then transfers to the original program. Typically, the user is unaware of the effect of the virus if the original program still does all that it used to. Most viruses attach in this manner.

Viruses That Surround a Program
An alternative to the attachment is a virus that runs the original program but has control before and after its execution. For example, a virus writer might want to prevent the virus from being detected. If the virus is stored on disk, its presence will be given away by its file name, or its size will affect the amount of space used on the disk. The virus writer might arrange for the virus to attach itself to the program that constructs the listing of files on the disk. If the virus regains control after the listing program has generated the listing but before the listing is displayed or printed, the virus could eliminate its entry from the listing and falsify space counts so that it appears not to exist.

Integrated Viruses and Replacements
A third situation occurs when the virus replaces some of its target, integrating itself into the original code of the target.. Clearly, the virus writer has to know the exact structure of the original program to know where to insert which pieces of the virus.

Virus Integrated into a Program.

Finally, the virus can replace the entire target, either mimicking the effect of the target or ignoring the expected effect of the target and performing only the virus effect. In this case, the user is most likely to perceive the loss of the original program.

Document Viruses

Currently, the most popular virus type is what we call the **document virus**, which is implemented within a formatted document, such as a written document, a database, a slide presentation, or a spreadsheet. These documents are highly structured files that contain both data (words or numbers) and commands (such as formulas, formatting controls, links). The commands are part of a rich programming language, including macros, variables and procedures, file accesses, and even system calls. The writer of a document virus uses any of the features of the programming language to perform malicious actions.

The ordinary user usually sees only the content of the document (its text or data), so the virus writer simply includes the virus in the commands part of the document, as in the integrated program virus.

How Viruses Gain Control

The virus (V) has to be invoked instead of the target (T). Essentially, the virus either has to seem to be T, saying effectively "I am T" (like some rock stars, where the target is the artiste formerly known as T) or the virus has to push T out of the way and become a substitute for T, saying effectively "Call me instead of T." A more blatant virus can simply say "invoke me [you fool]." The virus can assume T's name by replacing (or joining to) T's code in a file structure; this invocation technique is most appropriate for ordinary programs. The virus can overwrite T in storage (simply replacing the copy of T in storage, for example). Alternatively, the virus can change the pointers in the file table so that the virus is located instead of T whenever T is accessed through the file system.

Virus Completely Replacing a Program.

The virus can supplant T by altering the sequence that would have invoked T to now invoke the virus V; this invocation can be used to replace parts of the resident operating system by modifying pointers to those resident parts, such as the table of handlers for different kinds of interrupts.

Homes for Viruses

The virus writer may find these qualities appealing in a virus:

- It is hard to detect.
- It is not easily destroyed or deactivated.
- It spreads infection widely.
- It can reinfect its home program or other programs.
- It is easy to create.
- It is machine independent and operating system independent.

Few viruses meet all these criteria. The virus writer chooses from these objectives when deciding what the virus will do and where it will reside.

Just a few years ago, the challenge for the virus writer was to write code that would be executed repeatedly so that the virus could multiply. Now, however, one execution is enough to ensure widespread distribution. Many viruses are transmitted by e-mail, using either of two routes. In the first case, some virus writers generate a new e-mail message to all addresses in the victim's address book. These new messages contain a copy of the virus so that it propagates widely. Often the message is a brief, chatty, non-specific message that would encourage the new recipient to open the attachment from a friend (the first recipient). For example, the subject line or message body may read "I thought you might enjoy this picture from our vacation." In the second case, the virus writer can leave the infected file for the victim to forward unknowingly. If the virus's effect is not immediately obvious, the victim may pass the infected file unwittingly to other victims.

Let us look more closely at the issue of viral residence.

One-Time Execution
The majority of viruses today execute only once, spreading their infection and causing their effect in that one execution. A virus often arrives as an e-mail attachment of a document virus. It is executed just by being opened.

Boot Sector Viruses
A special case of virus attachment, but formerly a fairly popular one, is the so-called **boot sector virus.** When a computer is started, control begins with firmware that determines which hardware components are present, tests them, and transfers control to an operating system. A given hardware platform can run many different operating systems, so the operating system is not coded in firmware but is instead invoked dynamically, perhaps even by a user's choice, after the hardware test.

The operating system is software stored on disk. Code copies the operating system from disk to memory and transfers control to it; this copying is called the **bootstrap** (often **boot**) load because the operating system figuratively pulls itself into memory by its bootstraps. The firmware does its control transfer by reading a fixed number of bytes from a fixed location on the disk (called the **boot sector**) to a fixed address in memory and then jumping to that address (which will turn out to contain the first instruction of the bootstrap loader). The bootstrap loader then reads into memory the rest of the operating system from disk. To run a different operating system, the user just inserts a disk with the new operating system and a bootstrap loader. When the user reboots from this new disk, the loader there brings in and runs another operating system. This same scheme is used for personal computers, workstations, and large mainframes.

To allow for change, expansion, and uncertainty, hardware designers reserve a large amount of space for the bootstrap load. The boot sector on a PC is slightly less than 512 bytes, but since the loader will be larger than that, the hardware designers support "chaining," in which each block of the bootstrap is chained to (contains the disk location of) the next block. This chaining allows big bootstraps but also simplifies the installation of a virus. The virus writer simply breaks the chain at any point, inserts a pointer to the virus code to be executed, and reconnects the chain after the virus has been installed. This situation is shown in Figure.

Boot Sector Virus Relocating Code.

The boot sector is an especially appealing place to house a virus. The virus gains control very early in the boot process, before most detection tools are active, so that it can avoid, or at least complicate, detection. The files in the boot area are crucial parts of the operating system. Consequently, to keep users from accidentally modifying or deleting them with disastrous results, the operating system makes them "invisible" by not showing them as part of a normal listing of stored files, preventing their deletion. Thus, the virus code is not readily noticed by users.

Memory-Resident Viruses

Some parts of the operating system and most user programs execute, terminate, and disappear, with their space in memory being available for anything executed later. For very frequently used parts of the operating system and for a few specialized user programs, it would take too long to reload the program each time it was needed. Such code remains in memory and is called "resident" code. Examples of resident code are the routine that interprets keys pressed on the keyboard, the code that handles error conditions that arise during a program's execution, or a program that acts like an alarm clock, sounding a signal at a time the user determines. Resident routines are sometimes called TSRs or "terminate and stay resident" routines.

Virus writers also like to attach viruses to resident code because the resident code is activated many times while the machine is running. Each time the resident code runs, the virus does too. Once activated, the virus can look for and infect uninfected carriers. For example, after activation, a boot sector virus might attach itself to a piece of resident code. Then, each time the virus was activated it might check whether any removable disk in a disk drive was infected and, if not, infect it. In this way the virus could spread its infection to all removable disks used during the computing session.

Other Homes for Viruses

A virus that does not take up residence in one of these cozy establishments has to fend more for itself. But that is not to say that the virus will go homeless.

One popular home for a virus is an application program. Many applications, such as word processors and spreadsheets, have a "macro" feature, by which a user can record a series of commands and repeat them with one invocation. Such programs also provide a "startup macro" that is executed every time the application is executed. A virus writer can create a virus macro that adds itself to the startup directives for the application. It also then embeds a copy of itself in data files so that the infection spreads to anyone receiving one or more of those files.

Libraries are also excellent places for malicious code to reside. Because libraries are used by many programs, the code in them will have a broad effect. Additionally, libraries are often shared among users and transmitted from one user to another, a practice that spreads the infection. Finally, executing code in a library can pass on the viral infection to other transmission media. Compilers, loaders, linkers, runtime monitors, runtime debuggers, and even virus control programs are good candidates for hosting viruses because they are widely shared.

Virus Signatures
A virus cannot be completely invisible. Code must be stored somewhere, and the code must be in memory to execute. Moreover, the virus executes in a particular way, using certain methods to spread. Each of these characteristics yields a telltale pattern, called a **signature**, that can be found by a program that knows to look for it. The virus's signature is important for creating a program, called a **virus scanner**, that can automatically detect and, in some cases, remove viruses. The scanner searches memory and long-term storage, monitoring execution and watching for the telltale signatures of viruses. For example, a scanner looking for signs of the Code Red worm can look for a pattern containing the following characters:

```
/default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN

NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN

NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN

NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN

NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN

%u9090%u6858%ucbd3

%u7801%u9090%u6858%ucdb3%u7801%u9090%u6858

%ucbd3%u7801%u9090

%u9090%u8190%u00c3%u0003%ub00%u531b%u53ff

%u0078%u0000%u00=a

HTTP/1.0
```

When the scanner recognizes a known virus's pattern, it can then block the virus, inform the user, and deactivate or remove the virus. However, a virus scanner is effective only if it has been kept up-to-date with the latest information on current viruses. Side-bar 3-4 describes how viruses were the primary security breach among companies surveyed in 2001.

Storage Patterns

Most viruses attach to programs that are stored on media such as disks. The attached virus piece is invariant, so that the start of the virus code becomes a detectable signature. The attached piece is always located at the same position relative to its attached file. For example, the virus might always be at the beginning, 400 bytes from the top, or at the bottom of the infected file. Most likely, the virus will be at the beginning of the file, because the virus writer wants to obtain control of execution before the bona fide code of the infected program is in charge. In the simplest case, the virus code sits at the top of the program, and the entire virus does its malicious duty before the normal code is invoked. In other cases, the virus infection consists of only a handful of instructions that point or jump to other, more detailed instructions elsewhere. For example, the infected code may consist of condition testing and a jump or call to a separate virus module. In either case, the code to which control is transferred will also have a recognizable pattern.

Recognizable Patterns in Viruses.

A virus may attach itself to a file, in which case the file's size grows. Or the virus may obliterate all or part of the underlying program, in which case the program's size does not change but the program's functioning will be impaired. The virus writer has to choose one of these detectable effects.

The virus scanner can use a code or checksum to detect changes to a file. It can also look for suspicious patterns, such as a JUMP instruction as the first instruction of a system program (in case the virus has positioned itself at the bottom of the file but wants to be executed first).

Execution Patterns

A virus writer may want a virus to do several things at the same time, namely, spread infection, avoid detection, and cause harm. These goals are shown in Table 3-2, along with ways each goal can be addressed. Unfortunately, many of these behaviors are perfectly normal and might otherwise go undetected. For instance, one goal is modifying the file directory; many normal programs create files, delete files, and write to storage media. Thus, there are no key signals that point to the presence of a virus.

Most virus writers seek to avoid detection for themselves and their creations. Because a disk's boot sector is not visible to normal operations (for example, the contents of the boot sector do not show on a directory listing), many virus writers hide their code there. A resident virus can monitor disk accesses and fake the result of a disk operation that would show the virus hidden in a boot sector by showing the data that *should* have been in the boot sector (which the virus has moved elsewhere).

There are no limits to the harm a virus can cause. On the modest end, the virus might do nothing; some writers create viruses just to show they can do it. Or the virus can be relatively benign, displaying a message on the screen, sounding the buzzer, or playing music. From there, the problems can escalate. One virus can erase files, another an entire disk; one virus can prevent a computer from booting, and another can prevent writing to disk. The damage is bounded only by the creativity of the virus's author.

TABLE 3-2 Virus Effects and Causes.

Virus Effect	**How It Is Caused**
Attach to executable program	• Modify file directory • Write to executable program file
Attach to data or control file	• Modify directory • Rewrite data • Append to data • Append data to self
Remain in memory handler address table	• Intercept interrupt by modifying interrupt • Load self in non transient memory area
Infect disks	• Intercept interrupt • Intercept operating system call (to format disk, for example) • Modify system file • Modify ordinary executable program
Conceal self falsify result	• Intercept system calls that would reveal self and • Classify self as "hidden" file
Spread infection	• Infect boot sector • Infect systems program • Infect ordinary program • Infect data ordinary program reads to control its execution
Prevent deactivation de activation	• Activate before deactivating program and block • Store copy to reinfect after deactivation

Section 3.3 Viruses and Other Malicious Code

Transmission Patterns
A virus is effective only if it has some means of transmission from one location to another. As we have already seen, viruses can travel during the boot process, by attaching to an executable file or traveling within data files. The travel itself occurs during execution of an already infected program. Since a virus can execute any instructions a program can, virus travel is not confined to any single medium or execution pattern. For example, a virus can

arrive on a diskette or from a network connection, travel during its host's execution to a hard disk boot sector, reemerge next time the host computer is booted, and remain in memory to infect other diskettes as they are accessed.

Polymorphic Viruses

The virus signature may be the most reliable way for a virus scanner to identify a virus. If a particular virus always begins with the string 47F0F00E08 (in hexadecimal) and has string 00113FFF located at word 12, it is unlikely that other programs or data files will have these exact characteristics. For longer signatures, the probability of a correct match increases.

If the virus scanner will always look for those strings, then the clever virus writer can cause something other than those strings to be in those positions. For example, the virus could have two alternative but equivalent beginning words; after being installed, the virus will choose one of the two words for its initial word. Then, a virus scanner would have to look for both patterns. A virus that can change its appearance is called a **polymorphic virus.** (*Poly* means "many" and *morph* means "form".) A two-form polymorphic virus can be handled easily as two independent viruses. Therefore, the virus writer intent on preventing detection of the virus will want either a large or an unlimited number of forms so that the number of possible forms is too large for a virus scanner to search for. Simply embedding a random number or string at a fixed place in the executable version of a virus is not sufficient, because the signature of the virus is just the constant code excluding the random part. A polymorphic virus has to randomly reposition all parts of itself and randomly change all fixed data. Thus, instead of containing the fixed (and therefore searchable) string "HA! INFECTED BY A VIRUS," a polymorphic virus has to change even that pattern sometimes.

Trivially, assume a virus writer has 100 bytes of code and 50 bytes of data. To make two virus instances different, the writer might distribute the first version as 100 bytes of code followed by all 50 bytes of data. A second version could be 99 bytes of code, a jump instruction, 50 bytes of data, and the last byte of code. Other versions are 98 code bytes jumping to the last two, 97 and three, and so forth. Just by moving pieces around the virus writer can create enough different appearances to fool simple virus scanners. Once the scanner writers became aware of these kinds of tricks, however, they refined their signature definitions.

A more sophisticated polymorphic virus randomly intersperses harmless instructions throughout its code. Examples of harmless instructions include addition of zero to a number, movement of a data value to its own location, or a jump to the next instruction. These "extra" instructions make it more difficult to locate an invariant signature.

A simple variety of polymorphic virus uses encryption under various keys to make the stored form of the virus different. These are sometimes called **encrypting** viruses. This type of virus must contain three distinct parts: a decryption key, the (encrypted) object code of the virus, and the (unencrypted) object code of the decryption routine. For these viruses, the decryption routine itself or a call to a decryption library routine must be in the clear, and so that becomes the signature.

To avoid detection, not every copy of a polymorphic virus has to differ from every other copy. If the virus changes occasionally, not every copy will match a signature of every other copy.

The Source of Viruses
Since a virus can be rather small, its code can be "hidden" inside other larger and more complicated programs. Two hundred lines of a virus could be separated into one hundred packets of two lines of code and a jump each; these one hundred packets could be easily hidden inside a compiler, a database manager, a file manager, or some other large utility.

Virus discovery could be aided by a procedure to determine if two programs are equivalent. However, theoretical results in computing are very discouraging when it comes to the complexity of the equivalence problem. The general question, "are these two programs equivalent?" is undecidable (although that question *can* be answered for many specific pairs of programs). Even ignoring the general undecidability problem, two modules may produce subtly different results that may—or may not—be security relevant. One may run faster, or the first may use a temporary file for work space whereas the second performs all its computations in memory. These differences could be benign, or they could be a marker of an infection. Therefore, we are unlikely to develop a screening program that can separate infected modules from uninfected ones.

Although the general is dismaying, the particular is not. If we know that a particular virus may infect a computing system, we can check for it and detect it if it is there. Having found the virus, however, we are left with the task of cleansing the system of it. Removing the virus in a running system requires being able to detect and eliminate its instances faster than it can spread.

Prevention of Virus Infection
The only way to prevent the infection of a virus is not to share executable code with an infected source. This philosophy used to be easy to follow because it was easy to tell if a file was executable or not. For example, on PCs, a *.exe* extension was a clear sign that the file was executable. However, as we have noted, today's files are more complex, and a seemingly nonexecutable file may have some executable code buried deep within it. For example, a word processor may have commands within the document file; as we noted earlier, these commands, called macros, make it easy for the user to do complex or repetitive things. But they are really executable code embedded in the context of the document. Similarly, spreadsheets, presentation slides, and other office- or business-related files can contain code or scripts that can be executed in various ways—and thereby harbor viruses. And, as we have seen, the applications that run or use these files may try to be helpful by automatically invoking the executable code, whether you want it run or not! Against the principles of good security, e-mail handlers can be set to automatically open (without performing access control) attachments or embedded code for the recipient, so your e-mail message can have animated bears dancing across the top.

Another approach virus writers have used is a little-known feature in the Microsoft file design. Although a file with a *.doc* extension is expected to be a Word document, in fact, the true document type is hidden in a field at the start of the file. This convenience ostensibly helps a user who inadvertently names a Word document with a *.ppt* (Power-Point) or any other extension. In some cases, the operating system will try to open the associated application but, if that fails, the system will switch to the application of the hidden file type. So, the virus writer creates an executable file, names it with an inappropriate extension, and sends it to the victim, describing it is as a picture or a necessary code add-in or something else desirable. The unwitting recipient opens the file and, without intending to, executes the malicious code.

More recently, executable code has been hidden in files containing large data sets, such as pictures or read-only documents. These bits of viral code are not easily detected by virus scanners and certainly not by the human eye. For example, a file containing a photograph may be highly granular; if every sixteenth bit is part of a command string that can be executed, then the virus is very difficult to detect.

Since you cannot always know which sources are infected, you should assume that any outside source is infected. Fortunately, you know when you are receiving code from an outside source; unfortunately, it is not feasible to cut off all contact with the outside world.

In their interesting paper comparing computer virus transmission with human disease transmission, Kephart et al. observe that individuals' efforts to keep their computers free from viruses lead to communities that are generally free from viruses because members of the community have little (electronic) contact with the outside world. In this case, transmission is contained not because of limited contact but because of limited contact outside the community. Governments, for military or diplomatic secrets, often run disconnected network communities. The trick seems to be in choosing one's community prudently. However, as use of the Internet and the World Wide Web increases, such separation is almost impossible to maintain.

Nevertheless, there are several techniques for building a reasonably safe community for electronic contact, including the following:

- *Use only commercial software acquired from reliable, well-established vendors*. There is always a chance that you might receive a virus from a large manufacturer with a name everyone would recognize. However, such enterprises have significant reputations that could be seriously damaged by even one bad incident, so they go to some degree of trouble to keep their products virus-free and to patch any problem-causing code right away. Similarly, software distribution companies will be careful about products they handle.

- *Test all new software on an isolated computer.* If you must use software from a questionable source, test the software first on a computer with no hard disk, not connected to a network, and with the boot disk removed. Run the software and look for unexpected behavior, even simple behavior such as unexplained figures on the screen. Test the computer with a copy of an up-to-date virus scanner, created before running the suspect program. Only if the program passes these tests should it be installed on a less isolated machine.

- *Open attachments only when you know them to be safe.* What constitutes "safe" is up to you, as you have probably already learned in this chapter. Certainly, an attachment from an unknown source is of questionable safety. You might also distrust an attachment from a known source but with a peculiar message.

- *Make a recoverable system image and store it safely.* If your system does become infected, this clean version will let you reboot securely because it overwrites the corrupted system files with clean copies. For this reason, you must keep the image write- protected during reboot. Prepare this image now, before infection; after infection it is too late. For safety, prepare an extra copy of the safe boot image.

- *Make and retain backup copies of executable system files.* This way, in the event of a virus infection, you can remove infected files and reinstall from the clean backup copies (stored in a secure, offline location, of course).

- *Use virus detectors (often called virus scanners) regularly and update them daily.* Many of the virus detectors available can both detect and eliminate infection from viruses. Several scanners are better than one, because one may detect the viruses that others miss. Because scanners search for virus signatures, they are constantly being revised as new viruses are discovered. New virus signature files, or new versions of scanners, are distributed frequently; often, you can request automatic downloads from the vendor's web site. Keep your detector's signature file up-to-date.

Truths and Misconceptions About Viruses
Because viruses often have a dramatic impact on the computer-using community, they are often highlighted in the press, particularly in the business section. However, there is much misinformation in circulation about viruses. Let us examine some of the popular claims about them.

- *Viruses can infect only Microsoft Windows systems*. ***False***. Among students and office workers, PCs are popular computers, and there may be more people writing software (and viruses) for them than for any other kind of processor. Thus, the PC is most frequently the target when someone decides to write a virus. However, the principles of virus attachment and infection apply equally to other processors, including Macintosh computers, Unix workstations, and mainframe computers. In fact, no writeable stored- program computer is immune to possible virus attack. As we noted in Chapter 1, this situation means that *all* devices containing computer code, including automobiles, airplanes, microwave ovens, radios, televisions, and radiation therapy machines have the potential for being infected by a virus.

- *Viruses can modify "hidden" or "read only" files*. ***True***. We may try to protect files by using two operating system mechanisms. First, we can make a file a hidden file so that a user or program listing all files on a storage device will not see the file's name. Second, we can apply a read-only protection to the file so that the user cannot change the file's contents. However, each of these protections is applied by software, and virus software can override the native software's protection. Moreover, software protection is layered, with the operating system providing the most elementary protection. If a secure operating system obtains control *before* a virus contaminator has executed, the operating system can prevent contamination as long as it blocks the attacks the virus will make.

- *Viruses can appear only in data files, or only in Word documents, or only in programs*. ***False***. What are data? What is an executable file? The distinction between these two concepts is not always clear, because a data file can control how a program executes and even cause a program to execute. Sometimes a data file lists steps to be taken by the program that reads the data, and these steps can include executing a program. For example, some applications contain a configuration file whose data are exactly such steps. Similarly, word processing document files may contain startup commands to execute when the document is opened; these startup commands can contain malicious code. Although, strictly speaking, a virus can activate and spread only when a program executes, in fact, data files are acted upon by programs. Clever virus writers have been able to make data control files that cause programs to do many things, including pass along copies of the virus to other data files.

- *Viruses spread only on disks or only in e-mail****. False***. File-sharing is often done as one user provides a copy of a file to another user by writing the file on a transportable disk. However, any means of electronic file transfer will work. A file can be placed in a network's library or posted on a bulletin board. It can be attached to an electronic mail message or made available for download from a web site. Any mechanism for sharing files—of programs, data, documents, and so forth—can be used to transfer a virus.

- *Viruses cannot remain in memory after a complete power off/power on reboot*. ***True***. If a virus is resident in memory, the virus is lost when the memory loses power. That is, computer memory (RAM) is volatile, so that all contents are deleted when power is lost.[2] However, viruses written to disk certainly can remain through a reboot cycle and reappear after the reboot. Thus, you can receive a virus infection, the virus can be written to disk (or to network storage), you can turn the machine off and back on, and the virus can be reactivated during the reboot. Boot sector viruses gain control when a machine reboots (whether it is a hardware or software reboot), so a boot sector virus may remain through a reboot cycle because it activates immediately when a reboot has completed.

- *Viruses cannot infect hardware****. True***. Viruses can infect only things they can modify; memory, executable files, and data are the primary targets. If hardware contains writeable storage (so-called firmware) that can be accessed under program control, that storage *is* subject to virus attack. There have been a few

- *Viruses can be malevolent, benign, or benevolent*. ***True.*** Not all viruses are bad. For example, a virus might locate uninfected programs, compress them so that they occupy less memory, and insert a copy of a routine that decompresses the program when its execution begins. At the same time, the virus is spreading the compression function to other programs. This virus could substantially reduce the amount of storage required for stored programs, possibly by up to 50 percent. However, the compression would be done at the request of the virus, not at the request, or even knowledge, of the program owner.

2.4 TARGETED MALICIOUS PROGRAM

So far, we have looked at anonymous code written to affect users and machines indiscriminately. Another class of malicious code is written for a particular system, for a particular application, and for a particular purpose. Many of the virus writers' techniques apply, but there are also some new ones.

Trapdoors

A **trapdoor** is an undocumented entry point to a module. The trapdoor is inserted during code development, perhaps to test the module, to provide "hooks" by which to connect future modifications or enhancements or to allow access if the module should fail in the future. In addition to these legitimate uses, trapdoors can allow a programmer access to a program once it is placed in production.

Salami Attack

An attack known as a **salami attack**. This approach gets its name from the way odd bits of meat and fat are fused together in a sausage or salami. In the same way, a salami attack merges bits of seemingly inconsequential data to yield powerful results. For example, programs often disregard small amounts of money in their computations, as when there are fractional pennies as interest or tax is calculated.

Such programs may be subject to a salami attack, because the small amounts are shaved from each computation and accumulated elsewhere—such as the programmer's bank account! The shaved amount is so small that an individual case is unlikely to be noticed, and the accumulation can be done so that the books still balance overall. However, accumulated amounts can add up to a tidy sum, supporting a programmer's early retirement or new car. It is often the resulting expenditure, not the shaved amounts, that gets the attention of the authorities.

Covert Channels: Programs That Leak Information

So far, we have looked at malicious code that performs unwelcome actions. Next, we turn to programs that communicate information to people who should not receive it. The communication travels unnoticed, accompanying other, perfectly proper, communications. The general name for these extraordinary paths of communication is **covert channels**.

Suppose a group of students is preparing for an exam for which each question has four choices (a, b, c, d); one student in the group, Sophie, understands the material perfectly and she agrees to help the others. She says she will reveal the answers to the questions, in order, by coughing once for answer "a," sighing for answer "b," and so forth. Sophie uses a communications channel that outsiders may not notice; her communications are hidden in an open channel. This communication is a human example of a covert channel.

Timing Channels

Other covert channels, called **timing channels**, pass information by using the speed at which things happen. Actually, timing channels are shared resource channels in which the shared resource is time.

A service program uses a timing channel to communicate by using or not using an assigned amount of computing time. In the simple case, a multiprogrammed system with two user processes divides time into blocks and allocates blocks of processing alternately to one process and the other. A process is offered processing time, but if the process is waiting for another event to occur and has no processing to do, it rejects the offer. The service process either uses its block (to signal a 1) or rejects its block (to signal a 0).

2.5 CONTROL AGAINST PROGRAM THREAT

There are many ways a program can fail and many ways to turn the underlying faults into security failures. It is of course better to focus on prevention than cure; how do we use controls during software development—the specifying, designing, writing, and testing of the program—to find and eliminate the sorts of exposures we have discussed? The discipline of software engineering addresses this question more globally, devising approaches to ensure the quality of software. In this book, we provide an overview of several techniques that can prove useful in finding and fixing security flaws .

In this section we look at three types of controls: developmental, operating system, and administrative. We discuss each in turn.

Developmental Controls

Many controls can be applied during software development to ferret out and fix problems. So let us begin by looking at the nature of development itself, to see what tasks are involved in specifying, designing, building, and testing software. The Nature of Software Development Software development is often considered a solitary effort; a programmer sits with a specification or design and grinds out line after line of code. But in fact, software development is a collaborative effort, involving people with different skill sets who combine their expertise to produce a working product. Development requires people who can

- specify the system, by capturing the requirements and building a model of how the system should work from the users' point of view
- design the system, by proposing a solution to the problem described by the requirements and building a model of the solution
- implement the system, by using the design as a blueprint for building a working solution
- test the system, to ensure that it meets the requirements and implements the solution as called for in the design
- review the system at various stages, to make sure that the end products are consistent with the specification and design models
- document the system, so that users can be trained and supported
- manage the system, to estimate what resources will be needed for development and to track when the system will be done
- maintain the system, tracking problems found, changes needed, and changes made, and evaluating their effects on overall quality and functionality

One person could do all these things. But more often than not, a team of developers works together to perform these tasks. Sometimes a team member does more than one activity; a tester can take part in a requirements review, for example, or an implementer can write documentation. Each team is different, and team dynamics play a large role in the team's success.

We can examine both product and process to see how each contributes to quality and in particular to security as an aspect of quality. Let us begin with the product, to get a sense of how we recognize high quality secure software.

Modularity, Encapsulation, and Information Hiding

Code usually has a long shelf-life, and it is enhanced over time as needs change and faults are found and fixed. For this reason, a key principle of software engineering is to create a design or code in small, self-contained units, called components or modules; when a system is written this way, we say that it is modular. Modularity offers advantages for program development in general and security in particular.

If a component is isolated from the effects of other components, then it is easier to trace a problem to the fault that caused it and to limit the damage the fault causes. It is also easier to maintain the system, since changes to an isolated component do not affect other components. And it is easier to see where vulnerabilities may lie if the component is isolated. We call this isolation encapsulation.

Information hiding is another characteristic of modular software. When information is hidden, each component hides its precise implementation or some other design decision from the others. Thus, when a change is needed, the overall design can remain intact while only the necessary changes are made to particular components

2.6 PROTECTION IN GENERAL PURPOSE OPERATING SYSTEM PROTECTED OBJECT AND METHOD OF PROTECTION MEMORY AND ADDRESS PROTECTION

Protected objects

The rise of multiprogramming meant that several aspects of a computing system required protection:

- memory
- sharable I/O devices, such as disks
- serially reusable I/O devices, such as printers and tape drives
- sharable programs and subprocedures
- networks
- sharable data

As it assumed responsibility for controlled sharing, the operating system had to protect theseobjects.

Security in operating system

The basis of protection is separation: keeping one user's objects separate from other users.Rushby and Randell noted that separation in an operating system can occur in several ways:

- *physical separation*, in which different processes use different physical objects, such as separate printers for output requiring different levels of security
- *temporal separation*, in which processes having different security requirements are executed at different times
- *logical separation*, in which users operate under the illusion that no other processes exist, as when an operating system constrains a program's accesses so that the program cannot access objects outside its permitted domain
- *cryptographic separation*, in which processes conceal their data and computations in such a way that they are unintelligible to outside processes

Of course, combinations of two or more of these forms of separation are also possible.
The categories of separation are listed roughly in increasing order of complexity to implement, and, for the first three, in decreasing order of the security provided. However, the first two approaches are very stringent and can lead to poor resource utilization. Therefore, we would like to shift the burden of protection to the operating system to allow concurrent execution of processes having different security needs.
But separation is only half the answer. We want to separate users and their objects, but we also want to be able to provide sharing for some of those objects. For example, two users with different security levels may want to invoke the same search algorithm or function call.

We would like the users to be able to share the algorithms and functions without compromisingtheir individual security needs. An operating system can support separation and sharing in several ways, offering protection at any of several levels.

- *Do not protect.* Operating systems with no protection are appropriate when sensitive procedures are being run at separate times.
- *Isolate*. When an operating system provides isolation, different processes running concurrently are unaware of the presence of each other. Each process has its own address space, files, and other objects. The operating system must confine each process somehow so that the objects of the other processes are completely concealed.
- *Share all or share nothing.* With this form of protection, the owner of an objectdeclares it to be public or private. A public object is available to all users, whereas a private object is available only to its owner.
- *Share via access limitation.* With protection by access limitation, the operating system checks the allowability of each user's potential access to an object. That is, access control is implemented for a specific user and a specific object. Lists of acceptable actions guide the operating system in determining whether a particular user should have access to a particular object. In some sense, the operating system acts as a guard between users and objects, ensuring that only authorized accesses occur.
- *Share by capabilities.* An extension of limited access sharing, this form of protection allows dynamic creation of sharing rights for objects. The degree of sharing can depend on the owner or the subject, on the context of the computation, or on the object itself.
- *Limit use of an object.* This form of protection limits not just the access to an object but the use made of that object after it has been accessed. For example, a user may be allowed to view a sensitive document, but not to print a copy of it. More powerfully, a user may be allowed access to data in a database to derive statistical summaries (such as average salary at a particular grade level), but not to determine specific data values (salaries of individuals).

Methods of memory protection

Memory protection is a way to control memory access rights on a computer, and is a part of most modern operating systems. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug within a process from affecting other processes, or the operating system itself, and instead results in a segmentation fault or storage violation exception being sent to the offending process, generally causing abnormal termination (killing the process). Memory protection for computer security includes additional techniques such as address space layout randomization and executable space protection.

Segmentation

Segmentation refers to dividing a computer's memory into segments. A reference to a memory location includes a value that identifies a segment and an offset within that segment.The x86 architecture has multiple segmentation features, which are helpful for using protected memory on this architecture. On the x86 processor architecture, the Global Descriptor Table and Local Descriptor Tables can be used to reference segments in the computer's memory. Pointers to memory segments on x86 processors can also be stored in the processor's segment registers. Initially x86 processors had 4 segment registers, CS (code segment), SS (stack segment), DS (data segment) and ES (extra segment); later another two segment registers were added – FS and GS.

Paged virtual memory

In paging the memory address space is divided into equal-sized blocks called pages. Using virtual memory hardware, each page can reside in any location of the computer's physical memory, or be flagged as being protected. Virtual memory makes it possible to have a linear virtual memory address space and to use it to access blocks fragmented over physical memory address space.Most computer architectures which support paging also use pages as the basis for memory protection.

A *page table* maps virtual memory to physical memory. The page table is usually invisible to the process. Page tables make it easier to allocate additional memory, as each new page can be allocated from anywhere in physical memory.

It is impossible for an application to access a page that has not been explicitly allocated to it, because every memory address either points to a page allocated to that application, or generates an interrupt called a *page fault*. Unallocated pages, and pages allocated to any other application, do not have any addresses from the application point of view.

A page fault may not necessarily indicate an error. Page faults are not only used for memory protection. The operating system may manage the page table in such a way that a reference to a page that has been previously swapped out to disk causes a page fault. The operating system intercepts the page fault and, loads the required memory page, and the application continues as if no fault had occurred. This scheme, known as virtual memory, allows in- memory data not currently in use to be moved to disk storage and back in a way which is transparent to applications, to increase overall memory capacity.

software fault handler can, if desired, check the missing key against a larger list of keys maintained by software; thus, the protection key registers inside the processor may be treated as a software-managed cache of a larger list of keys associated with a process.

Simulated segmentation

Simulation is use of a monitoring program to interpret the machine code instructions of some computer architectures. Such an Instruction Set Simulator can provide memory protection by using a segmentation-like scheme and validating the target address and length of each instruction in real time before actually executing them. The simulator must calculate the target address and length and compare this against a list of valid address ranges that it holds concerning the thread's environment, such as any dynamic memoryblocks acquired since the thread's inception, plus any valid shared static memory slots. The meaning of "valid" may change throughout the thread's life depending upon context. It may sometimes be allowed to alter a static block of storage, and sometimes not, depending upon the current mode of execution, which may or may not depend on a storage key or supervisor state.

It is generally not advisable to use this method of memory protection where adequate facilities exist on a CPU, as this takes valuable processing power from the computer. However, it is generally used for debugging and testing purposes to provide an extra fine level of granularity to otherwise generic storage violations and can indicate precisely which instruction is attempting to overwrite the particular section of storage which may have the same storage key as unprotected storage.

2.7 FILE PROTECTION MECHANISM

Until now, we have examined approaches to protecting a general object, no matter the object's nature or type. But some protection schemes are particular to the type. To see how they work, we focus in this section on file protection. The examples we present are only representative; they do not cover all possible means of file protection on the market.

Basic Forms of Protection

We noted earlier that all multiuser operating systems must provide some minimal protection to keep one user from maliciously or inadvertently accessing or modifying the files of another. As the number of users has grown, so also has the complexity of these protection schemes.

All “None Protection

In the original IBM OS operating systems, files were by default public. Any user could read, modify, or delete a file belonging to any other user. Instead of software- or hardware-based protection, the principal protection involved trust combined with ignorance. System designers supposed that users could be trusted not to read or modify others' files, because the users would expect the same respect from others. Ignorance helped this situation, because a user could access a file only by name ; presumably users knew the names only of those files to which they had legitimate access.

However, it was acknowledged that certain system files were sensitive and that the system administrator could protect them with a password. A normal user could exercise this feature, but passwords were viewed as most valuable for protecting operating system files. Two philosophies guided password use. Sometimes, passwords were used to control all accesses (read, write, or delete), giving the system administrator complete control over all files. But at other times passwords would control only write and delete accesses , because only these two actions affected other users. In either case, the password mechanism required a system operator's intervention each time access to the filebegan .

However, this all-or-none protection is unacceptable for several reasons.

- Lack of trust . The assumption of trustworthy users is not necessarily justified. For systems with few users who all know each other, mutual respect might suffice; but in large systems where not every user knows every other user, there is no basis for trust.
- All or nothing . Even if a user identifies a set of trustworthy users, there is no convenient way to allow access only to them.
- Rise of timesharing . This protection scheme is more appropriate for a batch environment, in which users have little chance to interact with other users and in which users do their thinking and exploring when not interacting with the system. However, on timesharing systems, users interact with other users. Because users choose when to execute programs, they are more likely in a timesharing environment to arrange computing tasks to be able to pass results from one program or one user to another.
- Complexity . Because (human) operator intervention is required for this file protection, operating system performance is degraded. For this reason, this type of file protection is discouraged by computing centers for all but the most sensitive data sets.
- File listings . For accounting purposes and to help users remember for what files they are responsible, various system utilities can produce a list of all files. Thus, users are not

necessarily ignorant of what files reside on the system. Interactive users may try to browse through any unprotected files.

Group Protection

Because the all-or-nothing approach has so many drawbacks, researchers sought an improved way to protect files. They focused on identifying groups of users who had some common relationship. In a typical implementation, the world is divided into three classes: the user, a trusted working group associated with the user, and the rest of the users. For simplicity we can call these classes user, group, and world . This form of protection is used on some network systems and the Unix system.

All authorized users are separated into groups. A group may consist of several members working on a common project, a department, a class, or a single user. The basis for group membership is need to share . The group members have some common interest and therefore are assumed to have files to share with the other group members. In this approach, no user belongs to more than one group. (Otherwise, a member belonging to groups A and B could pass along an A file to another B group member.)

When creating a file, a user defines access rights to the file for the user, for other members of the same group, and for all other users in general. Typically, the choices for access rights are a limited set, such as {read, write, execute, delete}. For a particular file, a user might declare read-only access to the general world, read and write access to the group, and all rights to the user. This approach would be suitable for a paper being developed by a group, whereby the different members of the group might modify sections being written within the group. The paper itself should be available for people outside the group to review but not change.

A key advantage of the group protection approach is its ease of implementation. A user is recognized by two identifiers (usually numbers): a user ID and a group ID. These identifiers are stored in the file directory entry for each file and are obtained by the operating system when a user logs in. Therefore, the operating system can easily check whether a proposed access to a file is requested from someone whose group ID matches the group ID for the file to be accessed.

Although this protection scheme overcomes some of the shortcomings of the all-or-nothing scheme, it introduces some new difficulties of its own.

- Group affiliation . A single user cannot belong to two groups. Suppose Tom belongs to one group with Ann and to a second group with Bill. If Tom indicates that a file is to be readable by the group, to which group(s) does this permission refer? Suppose a file of Ann's is readable by the group; does Bill have access to it? These ambiguities are most simply resolved by declaring that every user belongs to exactly one group. (This restriction does not mean that all users belong to the same group.)
- Multiple personalities . To overcome the one-person one-group restriction, certain people might obtain multiple accounts, permitting them, in effect, to be multiple users. This hole in the protection approach leads to new problems, because a single person can be only one user at a time. To see how problems arise, suppose Tom obtains two accounts, thereby becoming Tom1 in a group with Ann and Tom2 in a group with Bill. Tom1 is not in the same group as Tom2, so any files, programs, or aids developed under the Tom1 account can be available to Tom2 only if they are available to the entire world. Multiple personalities lead to a proliferation of accounts, redundant files, limited protection for files of general interest, and inconvenience to users.
- All groups . To avoid multiple personalities, the system administrator may decide that Tom should have access to all his files any time he is active. This solution puts the responsibility

on Tom to control with whom he shares what things. For example, he may be in Group1 with Ann and Group2 with Bill. He creates a Group1 file to share with Ann. But if he is active in Group2 the next time he is logged in, he still sees the Group1 file and may not realize that it is not accessible to Bill, too.

- Limited sharing . Files can be shared only within groups or with the world. Users want to be able to identify sharing partners for a file on a per-file basis, for example, sharing one file with ten people and another file with twenty others.

Single Permissions

In spite of their drawbacks, the file protection schemes we have described are relatively simple and straightforward. The simplicity of implementing them suggests other easy-to- manage methods that provide finer degrees of security while associating permission with a single file.

Password or Other Token

We can apply a simplified form of password protection to file protection by allowing a user to assign a password to a file. User accesses are limited to those who can supply the correct password at the time the file is opened. The password can be required for any access or only for modifications (write access).

Password access creates for a user the effect of having a different "group" for every file. However, file passwords suffer from difficulties similar to those of authentication passwords:

- Loss . Depending on how the passwords are implemented, it is possible that no one will be able to replace a lost or forgotten password. The operators or system administrators can certainly intervene and unprotect or assign a particular password, but often they cannot determine what password a user has assigned; if the user loses the password, a new one must be assigned.
- Use . Supplying a password for each access to a file can be inconvenient and time consuming.
- Disclosure . If a password is disclosed to an unauthorized individual, the file becomes immediately accessible. If the user then changes the password to reprotect the file, all the other legitimate users must be informed of the new password because their old password will fail.
- Revocation . To revoke one user's access right to a file, someone must change the password, thereby causing the same problems as disclosure.

Temporary Acquired Permission

The Unix operating system provides an interesting permission scheme based on a three-level user “group “world hierarchy. The Unix designers added a permission called set use rid (suid) . If this protection is set for a file to be executed, the protection level is that of the file's owner , not the executor . To see how it works, suppose Tom owns a file and allows Ann to execute it with suid . When Ann executes the file, she has the protection rights of Tom, not of herself.

This peculiar-sounding permission has a useful application. It permits a user to establish data files to which access is allowed only through specified procedures.

For example, suppose you want to establish a computerized dating service that manipulates a database of people available on particular nights. Sue might be interested in a date for Saturday, but she might have already refused a request from Jeff, saying she had other plans. Sue instructs the service not to reveal to Jeff that she is available. To use the service, Sue, Jeff, and others must be able to read and write (at least indirectly) the file to determine who is available or to post their availability. But if Jeff can read the file directly, he would find that

Sue has lied. Therefore, your dating service must force Sue and Jeff (and all others) to access this file only through an access program that would screen the data Jeff obtains. But if the file access is limited to read and write by you as its owner, Sue and Jeff will never be able to enter data into it.

The solution is the Unix SUID protection. You create the database file, giving only you access permission. You also write the program that is to access the database, and save it with the SUID protection. Then, when Jeff executes your program, he temporarily acquires your access permission, but only during execution of the program. Jeff never has direct access to the file because your program will do the actual file access. When Jeff exits from your program, he regains his own access rights and loses yours. Thus, your program can access the file, but the program must display to Jeff only the data Jeff is allowed to see.

This mechanism is convenient for system functions that general users should be able to perform only in a prescribed way. For example, only the system should be able to modify the file of users' passwords, but individual users should be able to change their own passwords any time they wish. With the SUID feature, a password change program can be owned by the system, which will therefore have full access to the system password table. The program to change passwords also has SUID protection, so that when a normal user executes it, the program can modify the password file in a carefully constrained way on behalf of the user.

2.8 USER AUTHENTICATION

An operating system bases much of its protection on knowing who a user of the system is. In real-life situations, people commonly ask for identification from people they do not know: A bank employee may ask for a driver's license before cashing a check, library employees may require some identification before charging out books, and immigration officials ask for passports as proof of identity. In-person identification is usually easier than remote identification. For instance, some universities do not report grades over the telephone because the office workers do not necessarily know the students calling. However, a professor who recognizes the voice of a certain student can release that student's grades. Over time, organizations and systems have developed means of authentication, using documents, voice recognition, fingerprint and retina matching, and other trusted means of identification.
In computing, the choices are more limited and the possibilities less secure. Anyone can attempt to log in to a computing system. Unlike the professor who recognizes a student's voice, the computer cannot recognize electrical signals from one person as being any different from those of anyone else. Thus, most computing authentication systems must be based on some knowledge shared only by the computing system and the user. Authentication mechanisms use any of three qualities to confirm a user's identity.
1. Something the user *knows* a Passwords, PIN numbers, passphrases, a secret handshake, and mother's maiden name are examples of what a user may know.
2. Something the user *has* an Identity badge, physical keys, a driver's license, or a uniform are common examples of things people have that make them recognizable.
3. Something the user *is*a These authenticators, called biometrics, are based on a physical characteristic of the user, such as a fingerprint, the pattern of a person's voice, or a face (picture). These authentication methods are old (we recognize friends in person by their faces or on a telephone by their voices) but are just starting to be used in computer authentication.
Passwords as Authenticators
The most common authentication mechanism for user to operating system is a password, a

"word" known to computer and user. Although password protection seems to offer a relatively secure system, human practice sometimes degrades its quality. In this section we consider passwords, criteria for selecting them, and ways of using them for authentication. We conclude by noting other authentication techniques and by studying problems in the authentication process, notably Trojan horses masquerading as the computer authentication process.

Use of Passwords

Passwords are mutually agreed-upon code words, assumed to be known only to the user and the system. In some cases, a user chooses passwords; in other cases, the system assigns them. The length and format of the password also vary from one system to another.

Even though they are widely used, passwords suffer from some difficulties of use:

- Loss. Depending on how the passwords are implemented, it is possible that no one will be able to replace a lost or forgotten password. The operators or system administrators can certainly intervene and unprotect or assign a particular password, but often they cannot determine what password a user has chosen; if the user loses the password, a new one must be assigned.
- Use. Supplying a password for each access to a file can be inconvenient and time consuming.
- Disclosure. If a password is disclosed to an unauthorized individual, the file becomes immediately accessible. If the user then changes the password to reprotect the file, all other legitimate users must be informed of the new password because their old password will fail.
- Revocation. To revoke one user's access right to a file, someone must change the password, thereby causing the same problems as disclosure.

The use of passwords is fairly straightforward. A user enters some piece of identification, such as a name or an assigned user ID; this identification can be available to the public or easy to guess because it does not provide the real security of the system. The system then requests a password from the user. If the password matches that on file for the user, the user is authenticated and allowed access to the system. If the password match fails, the system requests the password again, in case the user mistyped.

2.9 DESIGNING TRUSTED O.S.

Operating systems are the prime providers of security in computing systems. They support many programming capabilities, permit multiprogramming and sharing of resources, and enforce restrictions on program and user behavior. Because they have such power, operating systems are also targets for attack, because breaking through the defence of an operating system gives access to the secrets of computing systems.

In we considered operating systems from the perspective of users, asking what primitive security services general operating systems provide. We studied these four services:

1. memory protection
2. file protection
3. general object access control
4. user authentication

We say that an operating system is trusted if we have confidence that it provides these four services consistently and effectively. In this chapter, we take the designer's perspective, viewing a trusted operating system in terms of the design and function of components that provide security services. The first four sections of this chapter correspond to the four major underpinnings of a trusted operating system:

1. Policy. Every system can be described by its requirements: statements of what the system should do and how it should do it. An operating system's security requirements are a set of well-defined, consistent, and implementable rules that have been clearly and unambiguously expressed. If the operating system is implemented to meet these requirements, it meets the user's expectations. To ensure that the requirements are clear, consistent, and effective, the operating system usually follows a stated security policy: a set of rules that lay out what is to be secured and why. We begin this chapter by studying several security policies for trusted operating systems.

2. Model. To create a trusted operating system, the designers must be confident that the proposed system will meet its requirements while protecting appropriate objects and relationships. They usually begin by constructing a model of the environment to be secured. The model is actually a representation of the policy the operating system will enforce. Designers compare the model with the system requirements to make sure that the overall system functions are not compromised or degraded by the security needs. Then, they study different ways of enforcing that security. In the second part of this chapter, we consider several different models for operating system security.

3. Design. After having selected a security model, designers choose a means to implement it. Thus, the design involves both what the trusted operating system is (that is, its intended functionality) and how it is to be constructed (its implementation). The third major section of this chapter addresses choices to be made during development of a trusted operating system.

4. Trust. Because the operating system plays a central role in enforcing security, we (as developers and users) seek some basis (assurance) for believing that it will meet our expectations. Our trust in the system is rooted in two aspects: features (the operating system has all the necessary functionality needed to enforce the expected security policy) and assurance (the operating system has been implemented in such a way that we have confidence it will enforce the security policy correctly and effectively). In the fourth part of this chapter, we explore what makes a particular design or implementation worthy of trust.

2.10 SECURITY POLICIES

To know that an operating system maintains the security we expect, we must be able to state its security policy. A security policy is a statement of the security we expect the system to enforce. An operating system (or any other piece of a trusted system) can be trusted only in relation to its security policy; that is, to the security needs the system is expected to satisfy.

Military Security Policy

Military security policy is based on protecting classified information. Each piece of information is ranked at a particular sensitivity level, such as *unclassified*, *restricted*, *confidential*, *secret*, or *top secret*. The ranks or levels form a hierarchy, and they reflect an increasing order of sensitivity.

Commercial Security Policies

Commercial enterprises have significant security concerns. They worry that industrial espionage will reveal information to competitors about new products under development. Likewise, corporations are often eager to protect information about the details of corporate finance. So even though the commercial world is usually less rigidly and less hierarchically

structured than the military world, we still find many of the same concepts in commercial security policies. For example, a large organization, such as a corporation or a university, may
be divided into groups or departments, each responsible for a number of disjoint projects. There may also be some corporate-level responsibilities, such as accounting and personnel activities. Data items at any level may have different degrees of sensitivity, such as *public*, *proprietary*, or *internal*; here, the names may vary among organizations, and no universal hierarchy applies.

2.11 MODELS OF SECURITY

In security and elsewhere, models are often used to describe, study, or analyze a particular situation or relationship. McLean gives a good overview of models for security. In particular, security models are used to

- test a particular policy for completeness and consistency
- document a policy help conceptualize and design an implementation
- check whether an implementation meets its requirements

We assume that some access control policy dictates whether a given user can access a particular object. We also assume that this policy is established outside any model. That is, a policy decision determines whether a specific user should have access to a specific object; the model is only a mechanism that enforces that policy. Thus, we begin studying models by considering simple ways to control access by one user.

Multilevel Security

Ideally, we want to build a model to represent a range of sensitivities and to reflect the need to separate subjects rigorously from objects to which they should not have access. For instance, consider an election and the sensitivity of data involved in the voting process. The names of the candidates are probably not sensitive. If the results have not yet been released,

the name of the winner is somewhat sensitive. If one candidate received an embarrassingly low number of votes, the vote count may be more sensitive. Finally, the way a particular individual voted is extremely sensitive. Users can also be ranked by the degree of sensitivity of information to which they can have access. For obvious reasons, the military has developed extensive procedures for securing information.

A generalization of the military model of information security has also been adopted as a model of data security within an operating system. Bell and La Padula [BEL73] were first to describe the properties of the military model in mathematical notation, and Denning first formalized the structure of this model. In 2005, Bell [BEL05] returned to the original model to highlight its contribution to computer security. He observed that the model demonstrated the need to understand security requirements before beginning system design, build security into not onto the system, develop a security toolbox, and design the system to protect itself. The generalized model is called the lattice model of security because its elements form a mathematical structure called a lattice. In this section, we describe the military example and then use it to explain the lattice model.

Lattice Model of Access Security

The military security model is representative of a more general scheme, called a lattice. The dominance relation defined in the military model is the relation for the lattice. The relations transitive and antisymmetric. The largest element of the lattice is the classification <topsecret; all compartments>, and the smallest element is <unclassified; no compartments>;these two elements respectively dominate and are dominated by all elements. Therefore, the military model is a lattice.

Many other structures are lattices. For example, we noted earlier that a commercial security policy may contain data sensitivities such as public, proprietary, and internal, with the natural ordering that public data are less sensitive than proprietary, which are less sensitive than internal. These three levels also form a lattice.

Many other structures are lattices. For example, we noted earlier that a commercial security policy may contain data sensitivities such as *public, proprietary,* and *internal,* with the natural ordering that *public* data are less sensitive than *proprietary*, which are less sensitive than *internal*. These three levels also form a lattice.
Security specialists have chosen to base security systems on a lattice because it naturally represents increasing degrees. A security system designed to implement lattice models can be used in a military environment. However, it can also be used in commercial environments with different labels for the degrees of sensitivity. Thus, lattice representation of sensitivity levels applies to many computing situations.

BellLaPadula Confidentiality Model
The Bell and La Padula model [BEL73] are a formal description of the allowable paths of information flow in a secure system. The model's goal is to identify allowable communication when maintaining secrecy is important. The model has been used to define security requirements for systems concurrently handling data at different sensitivity levels. This model is a formalization of the military security policy and was central to the U.S. Department of Defense's evaluation criteria, described later in this chapter.
We are interested in secure information flows because they describe acceptable connections between subjects and objects of different levels of sensitivity. One purpose for security- level analysis is to enable us to construct systems that can perform concurrent computation on data at two different sensitivity levels. For example, we may want to use one machine for top- secret and confidential data at the same time. The programs processing top-secret data would be prevented from leaking top-secret data to the confidential data, and the confidential users would be prevented from accessing the top-secret data. Thus, the BellLaPadula model is useful as the basis for the design of systems that handle data of multiple sensitivities.
To understand how the BellLaPadula model works, consider a security system with the following properties. The system covers a set of subjects S and a set of objects O. Each subject s in S and each object o in O has a fixed security class C(s) and C(o) (denoting clearance and classification level). The security classes are ordered by a relation . (Note: The classes may form a lattice, even though the BellLaPadula model can apply to even less restricted cases.)
Two properties characterize the secure flow of information.

Simple Security Property. A subject s may have read access to an object o only if $C(o) =< C(s)$.
In the military model, this property says that the security class (clearance) of someone receiving a piece of information must be at least as high as the class (classification) of the information.

***-Property** (called the "star property"). A subject s who has read access to an object o may have write access to an object p only if $C(o) =< C(p)$.

In the military model, this property says that the contents of a sensitive object can be written only to objects at least as high.
In the military model, one interpretation of the *-property is that a person obtaining information at one level may pass that information along only to people at levels no lower than the level of the information. The *-property prevents write-down, which occurs when a subject with access to high-level data transfers that data by writing it to a low-level object. Literally, the *-property requires that a person receiving information at one level not talk with people cleared at levels lower than the level of the information not even about the weather This example points out that this property is stronger than necessary to ensure security; the same is also true in computing systems. The BellLaPadula model is extremely conservative: It ensures security even at the expense of usability or other properties.

Biba Integrity Model
The BellLaPadula model applies only to secrecy of information: The model identifies paths that could lead to inappropriate *disclosure* of information. However, the integrity of data is important, too. Biba constructed a model for preventing inappropriate modification of data.

The Biba model is the counterpart (sometimes called the dual) of the BellLaPadula model. Bibadefines "integrity levels," which are analogous to the sensitivity levels of the BellLaPadulamodel. Subjects and objects are ordered by an integrity classification scheme, denoted $I(s)$ and $I(o)$. The properties are
Simple Integrity Property. Subject *s* can modify (have *write* access to) object *o* only if $I(s) >= I(o)$
Integrity *-Property. If subject *s* has *read* access to object *o* with integrity level $I(o)$, *s*can have *write* access to object *p* only if $I(o) >= I(p)$.

2.12 TRUSTED O.S. DESIGN

Operating systems by themselves (regardless of their security constraints) are very difficult to design. They handle many duties, are subject to interruptions and context switches, and must minimize overhead so as not to slow user computations and interactions. Adding the responsibility for security enforcement to the operating system substantially increases the difficulty of designing an operating system.

Nevertheless, the need for effective security is becoming more pervasive, and good software engineering principles tell us that it is better to design the security in at the beginning than to shoehorn it in at the end. Thus, this section focuses on the design of operating systems for a high degree of security. First, we examine the basic design of a standard multipurpose operating system. Then, we consider isolation, through which an operating system supports both sharing and separating user domains. We look in particular at the design of an operating system's kernel; how the kernel is designed suggests whether security will be provided effectively. We study two different interpretations of the kernel, and then we consider layered or ring-structured designs.

Trusted System Design Elements

That security considerations pervade the design and structure of operating systems implies two things. First, an operating system controls the interaction between subjects and objects, so security must be considered in every aspect of its design. That is, the operating system

design must include definitions of which objects will be protected in what way, which subjects will have access and at what levels, and so on. There must be a clear mapping from the security requirements to the design, so that all developers can see how the two relate. Moreover, once a section of the operating system has been designed, it must be checked to see that the degree of security that it is supposed to enforce or provide has actually been designed correctly. This checking can be done in many ways, including formal reviews or simulations.
Again, a mapping is necessary, this time from the requirements to design to tests so that developers can affirm that each aspect of operating system security has been tested and shown to work correctly.
Second, because security appears in every part of an operating system, its design and implementation cannot be left fuzzy or vague until the rest of the system is working and being tested. It is extremely hard to retrofit security features to an operating system designed with inadequate security. Leaving an operating system's security to the last minute is much like trying to install plumbing or wiring in a house whose foundation is set, structure defined, and walls already up and painted; not only must you destroy most of what you have built, but you may also find that the general structure can no longer accommodate all that is needed (and so some has to be left out or compromised). Thus, security must be an essential part of the initial design of a trusted operating system. Indeed, the security considerations may shape many of the other design decisions, especially for a system with complex and constraining security requirements. For the same reasons, the security and other design principles must be carried throughout implementation, testing, and maintenance.
Good design principles are always good for security, as we have noted above. But several important design principles are quite particular to security and essential for building a solid, trusted operating system. These principles have been articulated well by Saltzer and Saltzer and Schroeder :
 Least privilege. Each user and each program should operate by using the fewest privileges possible. In this way, the damage from an inadvertent or malicious attack is minimized.
 Economy of mechanism. The design of the protection system should be small, simple, and straightforward. Such a protection system can be carefully analyzed, exhaustively tested, perhaps verified, and relied on.
 Open design. The protection mechanism must not depend on the ignorance of potential attackers; the mechanism should be public, depending on secrecy of relatively few key items, such as a password table. An open design is also available for extensive public scrutiny, thereby providing independent confirmation of the design security.
 Complete mediation. Every access attempt must be checked. Both direct access attempts (requests) and attempts to circumvent the access checking mechanism should be considered, and the mechanism should be positioned so that it cannot be circumvented.
 Permission based. The default condition should be denial of access. A conservative designer identifies the items that *should* be accessible, rather than those that should *not.*
 Separation of privilege. Ideally, access to objects should depend on more than one condition, such as user authentication plus a cryptographic key. In this way, someone who defeats one protection system will not have complete access.
 Least common mechanism. Shared objects provide potential channels for information flow. Systems employing physical or logical separation reduce the risk from sharing.
 Ease of use. If a protection mechanism is easy to use, it is unlikely to be avoided.

Although these design principles were suggested several decades ago, they are as accurate now as they were when originally written. The principles have been used repeatedly and successfully in the design and implementation of numerous trusted systems. More importantly, when security problems have been found in operating systems in the past, they almost always derive from failure to abide by one or more of these principles.

2.13 ASSURANCE IN TRUSTED O.S.

Typical Operating System Flaws

Periodically throughout our analysis of operating system security features, we have used the phrase "exploit a vulnerability." Throughout the years, many vulnerabilities have been uncovered in many operating systems. They have gradually been corrected, and the body of knowledge about likely weak spots has grown.

Known Vulnerabilities
In this section, we discuss typical vulnerabilities that have been uncovered in operating systems. Our goal is not to provide a "how-to" guide for potential penetrators of operating systems. Rather, we study these flaws to understand the careful analysis necessary in designing and testing operating systems. User interaction is the largest single source of operating system vulnerabilities, for several reasons:
 The user interface is performed by independent, intelligent hardware subsystems. The human computer interface often falls outside the security kernel or security restrictions implemented by an operating system.
 Code to interact with users is often much more complex and much more dependent on the specific device hardware than code for any other component of the computing system. For these reasons, it is harder to review this code for correctness, let alone to verify it formally.
 User interactions are often character oriented. Again, in the interest of fast data transfer, the operating systems designers may have tried to take shortcuts by limiting the number of instructions executed by the operating system during actual data transfer. Sometimes the instructions eliminated are those that enforce security policies as each character is transferred.
A second prominent weakness in operating system security reflects an ambiguity in access policy. On one hand, we want to separate users and protect their individual resources. On the other hand, users depend on shared access to libraries, utility programs, common data, and system tables. The distinction between isolation and sharing is not always clear at the policy level, so the distinction cannot be sharply drawn at implementation.

A third potential problem area is incomplete mediation. Recall that Saltzer recommended an operating system design in which every requested access was checked for proper authorization. However, some systems check access only once per user interface operation, process execution, or machine interval. The mechanism is available to implement full protection, but the policy decision on when to invoke the mechanism is not complete.

Therefore, in the absence of any explicit requirement, system designers adopt the "most efficient" enforcement; that is, the one that will lead to the least use of machine resources.

Generality is a fourth protection weakness, especially among commercial operating systems for large computing systems. Implementers try to provide a means for users to customize their operating system installation and to allow installation of software packages written by other companies. Some of these packages, which themselves operate as part of the operating system, must execute with the same access privileges as the operating system. For example, there are programs that provide stricter access control than the standard control available from the operating system. The "hooks" by which these packages are installed are also trapdoors for any user to penetrate the operating system. Thus, several well-known points of security weakness are common to many commercial operating systems. Let us consider several examples of actual vulnerabilities that have been exploited to penetrate operating systems.

Testing

Testing is the most widely accepted assurance technique. As Boebert observes, conclusions from testing are based on the actual product being evaluated, not on some abstraction or precursor of the product. This realism is a security advantage. However, conclusions based on testing are necessarily limited, for the following reasons:

- Testing can demonstrate the existence of a problem, but passing tests does not demonstrate the absence of problems.
- Testing adequately within reasonable time or effort is difficult because the combinatorial explosion of inputs and internal states makes testing very complex.
- Testing based only on observable effects, not on the internal structure of a product, does not ensure any degree of completeness.
- Testing based on the internal structure of a product involves modifying the product by adding code to extract and display internal states. That extra functionality affects the product's behavior and can itself be a source of vulnerabilities or mask other vulnerabilities.
- Testing real-time or complex systems presents the problem of keeping track of all states and triggers. This problem makes it hard to reproduce and analyze problems reported as testers proceed.

Formal Verification

The most rigorous method of analyzing security is through formal verification, which was introduced in Chapter 3. Formal verification uses rules of mathematical logic to demonstrate that a system has certain security properties. In formal verification, the operating system is modeled and the operating system principles are described as assertions. The collection of models and assertions is viewed as a theorem, which is then proven. The theorem asserts that the operating system is correct. That is, formal verification confirms that the operating system provides the security features it should and nothing else.
Proving correctness of an entire operating system is a formidable task, often requiring months or even years of effort by several people. Computer programs called theorem provers can assist in this effort, although much human activity is still needed.

Validation

Formal verification is a particular instance of the more general approach to assuring correctness: verification. Validation is the counterpart to verification, assuring that the system developers have implemented all requirements. Thus, validation makes sure that the developer is building the right product (according to the specification), and verification checks the quality of the implementation . There are several different ways to validate an operating system.

 Requirements checking. One technique is to cross-check each operating system requirement with the system's source code or execution-time behavior. The goal is to demonstrate that the system does each thing listed in the functional requirements. This process is a narrow one, in the sense that it demonstrates only that the system does everything it should do. In security, we are equally concerned about prevention: making sure the system does not do the things it is not supposed to do. Requirements checking seldom addresses this aspect of requirements compliance.

 Design and code reviews. Design and code reviews usually address system correctness (that is, verification). But a review can also address requirements implementation. To support validation, the reviewers scrutinize the design or the code to ensure traceability from each requirement to design and code components, noting problems along the way (including faults, incorrect assumptions, incomplete or inconsistent behavior, or faulty logic). The success of this process depends on the rigor of the review.

 System testing. The programmers or an independent test team select data to check the system. These test data can be organized much like acceptance testing, so behaviors and data expected from reading the requirements document can be confirmed in the actual running of the system. The checking is done in a methodical manner to ensure completeness.

2.14 DIGITAL SIGNATURE

A **digital signature** is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, such that the sender cannot deny having sent the message (authentication and non-repudiation) and that the message was not altered in transit (integrity). Digital signatures are commonly used for software distribution, financial transactions, and in other cases where it is important to detect forgery or tampering.

Digital signatures are often used to implement electronic signatures, a broader term that refers to any electronic data that carries the intent of a signature, but not all electronic signatures use digital signatures. In some countries, including the United States, India, Brazil, and members of the European Union, electronic signatures have legal significance.

Digital signatures employ a type of asymmetric cryptography. For messages sent through a nonsecure channel, a properly implemented digital signature gives the receiver reason to believe the message was sent by the claimed sender. In many instances, common with Engineering companies for example, digital seals are also required for another layer of validation and security. Digital seals and signatures are equivalent to handwritten signatures and stamped seals.[5] Digital signatures are equivalent to traditional handwritten signatures in many respects, but properly implemented digital signatures are more difficult to forge than the handwritten type. Digital signature schemes, in the sense used here, are cryptographically based, and must be implemented properly to be effective. Digital signatures can also provide non-repudiation, meaning that the signer cannot successfully claim they did not sign a message, while also claiming their private key remains secret; further, some non- repudiation schemes offer a time stamp for the digital signature, so that even if the private key is exposed, the signature is valid. Digitally signed messages may be anything representable as a bitstring: examples include electronic mail, contracts, or a message sent via some other cryptographic protocol.

We are all familiar with the concept of a signature. We sign a document to show that it originated from us or was approved by us. The signature is proof to the recipient that the document comes from the correct entity. When a customer signs a check to himself, the bank needs to be sure that the check is issued by that customer and nobody else. In other words, a signature on a document, when verified, is a sign of authentication; the document is authentic. Consider a painting signed by an artist. The signature on the art, if authentic, means that the painting is probably authentic.

When Alice sends a message to Bob, Bob needs to check the authenticity of the sender; he needs to be sure that the message comes from Alice and not Eve. Bob can ask Alice to sign the message electronically. In other words, an electronic signature can prove the authenticity of Alice as the sender of the message. We refer to this type of signature as a digital signature.

Comparison

Before we continue any further, let us discuss the differences between two types of signatures:

conventional and digital.

Inclusion

A conventional signature is included in the document; it is part of the document. When we write a check, the signature is on the check; it is not a separate document. On the other hand, when we sign a document digitally, we send the signature as a separate document. The sender sends two documents: the message and the signature. The recipient receives both documents and verifies that the signature belongs to the supposed sender. If this is proved, the message is kept; otherwise, it is rejected.

Verification Method

The second difference between the two types of documents is the method of verifying the signature. In conventional signature, when the recipient receives a document, she compares the signature on the document with the signature on file. If they are the same, the document is authentic. The recipient needs to have a copy of this signature on file for comparison. In digital signature, the recipient receives the message and the signature. A copy of the signature is not stored anywhere. The recipient needs to apply a verification technique to the combination of the message and the signature to verify the authenticity.

Relationship

In conventional signature, there is normally a one-to-many relationship between a signature and documents. A person, for example, has a signature that is used to sign many checks, many documents, etc. In digital signature, there is a one-to-one relationship between a signature and a message. Each message has its own signature. The signature of one message cannot be used in another message. If Bob receives two messages, one after another, from Alice, he cannot use the signature of the first message to verify the second. Each message needs a new signature.

Duplicity

Another difference between the two types of signatures is a quality called duplicity. In conventional signature, a copy of the signed document can be distinguished from the original one on file. In digital signature, there is no such distinction unless there is a factor of time (such as a timestamp) on the document. For example, suppose Alice sends a document instructing Bob to pay Eve. If Eve intercepts the document and the signature, she can resend it later to get money again from Bob.

Need for Keys

In conventional signature a signature is like a private "key" belonging to the signer of the document. The signer uses it to sign a document; no one else has this signature. The copy of the signature is on file like a public key; anyone can use it to verify a document, to compare it to the original signature.

In digital signature, the signer uses her private key, applied to a signing algorithm, to sign the document. The verifier, on the other hand, uses the public key of the signer, applied to the verifying algorithm, to verify the document. Can we use a secret (symmetric) key to both sign and verify a signature? The answer is no for several reasons. First, a secret key is known only between two entities (Alice and Bob, for example). So, if Alice needs to sign another document and send it to Ted, she needs to use another secret key. Second, as we will see, creating a secret key for a session involves authentication, which normally uses digital signature. We have a vicious cycle. Third, Bob could use the secret key between himself and Alice, sign a document, send it to Ted, and pretend that it came from Alice.

Process

Digital signature can be achieved in two ways: signing the document or signing a digest of the document.

Signing the Document

Probably, the easier, but less efficient way is to sign the document itself. Signing a document is encrypting it with the private key of the sender; verifying the document is decrypting it with the public key of the sender.

We should make a distinction between private and public keys as used in digital signature and public and private keys as used for confidentiality. In the latter, the private and public keys of the receiver are used in the process. The sender uses the public key of the receiver to encrypt; the receiver uses his own private key to decrypt. In digital signature, the private and public keys of the sender are used. The sender uses her private key; the receiver uses the public key of the sender.

Signing the Digest

We mentioned that the public key is very inefficient in a cryptosystem if we are dealing with long messages. In a digital signature system, our messages are normally long, but we have to use public keys. The solution is not to sign the message itself; instead, we sign a

digest of the message. As we learned, a carefully selected message digest has a one-to-one relationship with the message. The sender can sign the message digest, and the receiver can verify the message digest. The effect is the same.
A digest is made out of the message at Alice's site. The digest then goes through the signing process using Alice's private key. Alice then sends the message and the signature to Bob.
At Bob's site, using the same public hash function, a digest is first created out of the received message. Calculations are done on the signature and the digest. The verifying process also applies criteria on the result of the calculation to determine the authenticity of the signature. If authentic, the message is accepted; otherwise, it is rejected.

Services

A digital signature can provide three services: message integrity, message authentication, and nonrepudiation. Note that a digital signature scheme does not provide confidential communication. If confidentiality is required, the message and the signature must be encrypted using either a secret-key or public-key cryptosystem.

Message Integrity

The integrity of the message is preserved even if we sign the whole message because we cannot get the same signature if the message is changed. The signature schemes today use a hash function in the signing and verifying algorithms that preserve the integrity of the message.

Message Authentication

A secure signature scheme, like a secure conventional signature (one that cannot be easily copied), can provide message authentication. Bob can verify that the message is sent by Alice because Alice's public key is used in verification. Alice's public key cannot create the same signature as Eve's private key.

Message Nonrepudiation

If Alice signs a message and then denies it, can Bob later prove that Alice actually signed it? For example, if Alice sends a message to a bank (Bob) and asks to transfer $10,000 from her account to Ted's account, can Alice later deny that she sent this message? With the scheme we have presented so far, Bob might have a problem. Bob must keep the signature on file and later use Alice's public key to create the original message to prove the message in the file and the newly created message are the same. This is not feasible because Alice may have changed her private/public key during this time; she may also claim that the file containing the signature is not authentic.

2.15 AUTHENTICATION

In the context of computer systems, authentication is a process that ensures and confirms a user’s identity. Authentication is one of the five pillars of information assurance (IA). The other four are integrity, availability, confidentiality and nonrepudiation.
Authentication begins when a user tries to access information. First, the user must prove his access rights and identity. When logging into a computer, users commonly enter usernames and passwords for authentication purposes. This login combination, which must be assigned to each user, authenticates access. However, this type of authentication can be circumvented by hackers.

A better form of authentication, biometrics, depends on the user’s presence and biological makeup (i.e., retina or fingerprints). This technology makes it more difficult for hackers to break into computer systems.

The Public Key Infrastructure (PKI) authentication method uses digital certificates to prove a user’s identity. There are other authentication tools, too, such as key cards and USB tokens. One of the greatest authentication threats occurs with email, where authenticity is often difficult to verify. For example, unsecured emails often appear legitimate.

2.16 SECRET SHARING

Secret sharing (also called **secret splitting**) refers to methods for distributing a *secret* amongst a group of participants, each of whom is allocated a *share* of the secret. The secret can be reconstructed only when a sufficient number, of possibly different types, of shares are combined together; individual shares are of no use on their own.

In one type of secret sharing scheme there is one *dealer* and *n players*. The dealer gives a share of the secret to the players, but only when specific conditions are fulfilled will the players be able to reconstruct the secret from their shares. The dealer accomplishes this by giving each player a share in such a way that any group of *t* (for *threshold*) or more players can together reconstruct the secret but no group of fewer Than*t* players can. Such a system is called a *(t, n)*-threshold scheme (sometimes it is written as an *(n, t)*-threshold scheme).

Importance of secure sharing

Secret sharing schemes are ideal for storing information that is highly sensitive and highly important. Examples include: encryption keys, missile launch codes, and numbered bank accounts. Each of these pieces of information must be kept highly confidential, as their exposure could be disastrous, however, it is also critical that they should not be lost. Traditional methods for encryption are ill-suited for simultaneously achieving high levels of confidentiality and reliability. This is because when storing the encryption key, one must choose between keeping a single copy of the key in one location for maximum secrecy, or keeping multiple copies of the key in different locations for greater reliability. Increasing reliability of the key by storing multiple copies lowers confidentiality by creating additional attack vectors; there are more opportunities for a copy to fall into the wrong hands. Secret sharing schemes address this problem, and allow arbitrarily high levels of confidentiality and reliability to be achieved.

Secure vs Insecure sharing

A secure secret sharing scheme distributes shares so that anyone with fewer than *t* shares have no extra information about the secret than someone with 0 shares.

Consider for example the secret sharing scheme in which the secret phrase "password" is divided into the shares "pa------," "--ss----," "----wo--," and "   rd,". A person with 0 shares knows only that the password consists of eight letters. He would have to guess the password from 26^8 = 208 billion possible combinations. A person with one share, however, would have to guess only the six letters, from 26^6 = 308 million combinations, and so on as more persons collude. Consequently, this system is not a "secure" secret sharing scheme, because a player with fewer than *t* secret-shares is able to reduce the problem of obtaining the inner secret without first needing to obtain all of the necessary shares.

In contrast, consider the secret sharing scheme where X is the secret to be shared, P_I are public asymmetric encryption keys and Q_I their corresponding private keys. Each player J is provided with $\{P_1(P_2(...(P_N(X)))), Q_j\}$. In this scheme, any player with a private key 1 can remove the outer layer of encryption, a player with keys 1 and 2 can remove the first and second layer, and so on. A player with fewer than N keys can never fully reach the secret X without first needing to decrypt a public-key-encrypted blob for which he does not have the corresponding private key - a problem that is currently believed to be computationally infeasible. Additionally, we can see that any user with all N private keys is able to decrypt all of the outer layers to obtain X, the secret, and consequently this system is a secure secret distribution system.

2.17 GROUP-ORIENTED CRYPTOGRAPHY

We say that an encryption scheme is group-oriented if the parties involved in encryption and decryption are more than two in number. To date, many group-oriented encryption applications have been addressed. In the following, we review well-known applications that have appeared in the literature.

1.Broadcast encryption. Consider the problem of broadcasting digital contents to a large set of authorized users. Such applications include paid-TV systems, copyrighted CD/DVD distributions, and fee-based online databases. The problem is that anyone connected to a broadcast channel is able to pick up the data, whether they are authorized or not. To prevent unauthorized users from extracting data, the broadcaster encrypts the message and only the authorized users have the decryption keys to recover the data. However, the proposed method carries out *n* encryptions for each copy of data, where *n* is the number of subscribers. \

2. Traitor tracing. In broadcast encryption, malicious authorized users, called traitors, may use their personal decryption keys to create a pirate decoder. The resulting pirate decoder allows an unauthorized user to extract the context. To discourage authorized users from revealing their keys, traitor tracing is first introduced by Chor, et al. The idea is an algorithm that uses the confiscated pirate decoder to track down at least one colluder without wrongly accusing non colluders with high probability. Most of these traitor-tracing schemes use a secret-key encryption scheme to encrypt data. A public-key traitor tracing allows everyone to perform encryption, and thus anyone can broadcast messages to authorized users securely.

3.Threshold cryptosystems. Within a group, various access policies are possible. Depending on the internal organization of the group and the access type of the message imposed by the sender, a different cryptographic scheme with the corresponding key management policy is needed. Threshold cryptosystems allow one to send encrypted messages to a group, while only a group achieving a “threshold" has the ability to reconstruct the plaintext.

2.18 IDENTIFICATION

Identification and Authentication(I&A) is the process of verifying that an identity is bound to the entity that makes an assertion or claim of identity. The I&A process assumes that there was an initial validation of the identity, commonly called identity proofing. Various methods of identity proofing are available, ranging from in-person validation using government issued identification, to anonymous methods that allow the claimant to remain anonymous, but known to the system if they return. The method used for identity proofing and validation should provide an assurance level commensurate with the intended use of the identity within the system. Subsequently, the entity asserts an identity together with an authenticator as a means for validation. The only requirements for the identifier are that it must be unique within its security domain.

Authenticators are commonly based on at least one of the following four factors:

- *Something you know*, such as a password or a personal identification number (PIN). This assumes that only the owner of the account knows the password or PIN needed to access the account.
- *Something you have*, such as a smart card or security token. This assumes that only the owner of the account has the necessary smart card or token needed to unlock the account.
- *Something you are*, such as fingerprint, voice, retina, or iris characteristics.
- *Where you are*, for example inside or outside a company firewall, or proximity of login location to a personal GPS device.

MODULE 3

THREATS IN NETWORK

Main aims of threats are to compromise confidentiality, integrity applied against data, software, hardware by nature accidents, non-malicious humans and malicious attackers.

What Makes a Network Vulnerable?

1. *Anonymity*
2. *Many Points of Attack*
3. *Sharing*
4. *Complexity Of System*

Threat Precursors:

1. Port scan
2. Social Engineering
3. Reconnaissance
4. Operating System and Application fingerprinting
5. Bulletin Boards and chats
6. Availability of Documentation

Threats In Transit: Eavesdropping and Wiretapping

The term **eavesdrop** implies overhearing without expanding any extra effort. For example, we can say that an attacker is eavesdropping by monitoring all traffic passing through a node.

The more hostile term is **wiretap**, which means intercepting communication through some effort.

Choices of wiretapping are:

1. Cable
2. Microwave
3. Satellite Communication
4. Optical Fiber
5. Wireless

From, a security stand point we should assume all communication links between network nodes that can broke. For this reason, commercial network users employ encryption to protect the confidentiality of their communication.

Protocol Flaws:

Each protocol is identified by its Request for Comment (RFC) number. In TCP, the sequence number of the client increments regularly which can be easily guessed and also which will be the next number.

Impersonation:

In many instances, there is an easier way than wiretapping for obtaining information on a network: impersonate another person or process.

In impersonation, an attacker has several choices:

- Guess the identity and authentication details of the target
- Disable authentication mechanism at the target computer
- Use a target that will not be authenticated
- Use a target whose authentication data are known

Spoofing:

Obtaining the network authentication credentials of an entity(a user, an account, a process, a node, a device) permits an attacker to create a full communication under the entity's identity. Examples of spoofing are masquerading, session hijacking, and man-in-the-middle attacks.

- In a masquerade one host pretends to be another.
- Session hijacking is intercepting and carrying on a session begun by another entity.
- Man-in-the-middle attack is a similar form of attack, in which one entity intrudes between two others.

Message Confidentiality Threats:

An attacker can easily violate message confidentiality (and perhaps integrity) because of the public nature of networks. Eavesdropping and impersonation attacks can lead to a confidentiality or integrity failure. Here we consider several other vulnerabilities that can affect confidentiality.

1. Mis delivery
2. Exposure
3. Traffic Flow Analysis

Message Integrity Threats:

In many cases, the *integrity* or correctness of a communication is at least as important as its confidentiality. In fact, for some situations, such as passing authentication data, the integrity of the communication is paramount. Threats based upon failures of integrity in communication

- Falsification of messages
- Noise

Web Site Defacement:

One of the most widely known attacks is the web site defacement attack. Because of the large number of sites that have been defaced and the visibility of the result, the attacks are often reported in the popular press. A defacement is common not only because of its visibility but also because of the ease with which one can be done.

The website vulnerabilities enable attacks known as buffer overflows, dot-dot problems, application code errors, and server side include problems.

Denial of Service:

Availability attacks, sometimes called denial-of-service or DOS attacks, are much more significant in networks than in other contexts. There are many accidental and malicious threats to availability or continued service. There are many accidental and malicious threats to availability or continued service.

1) Transmission Failure
2) Connection Flooding
3) Echo-Chargen
4) Ping of Death
5) Smurf
6) Syn Flood
7) Teardrop
8) Traffic Redirection
9) DNS Attacks

Threats in Active or Mobile Code:

Active code or mobile code is a general name for code that is pushed to the client for execution. Why should the web server waste its precious cycles and bandwidth doing simple work that the client's workstation can do? For example, suppose you want your web site to have bears dancing across the top of the page. To download the dancing bears, you could download a new image for each movement the bears take: one bit forward, two bits forward, and so forth. However, this approach uses far too much server time and bandwidth to compute the positions and download new images. A more efficient use of (server) resources is to download a program that runs on the client's machine and implements the movement of the bears.

Network Security Controls

The list of security attacks is long, and the news media carry frequent accounts of serious security incidents.

Security Threat Analysis:

The three steps of a security threat analysis in other situations are described here. First, we scrutinize all the parts of a system so that we know what each part does and how it interacts with other parts. Next, we consider possible damage to confidentiality, integrity, and availability. Finally, we hypothesize the kinds of attacks that could cause this damage. We can take the same steps with a network. We begin by looking at the individual parts of a network:

All the threats are summarized with a list as

- Intercepting data in traffic
- Accessing programs or data at remote hosts
- Modifying programs or data at remote hosts
- Modifying data in transit
- Inserting communications
- Impersonating a user
- Inserting a repeat of a previous communication
- Blocking selected traffic
- Blocking all traffic
- Running a program at a remote host

Design and Implementation:

Architecture:

As with so many of the areas we have studied, planning can be the strongest control. In particular, when we build or modify computer-based systems, we can give some thought to their overall architecture and plan to "build in" security as one of the key constructs. Similarly, the architecture or design of a network can have a significant effect on its security. The main areas to cover are

- Segmentation
- Redundancy
- Single point of failure
- Mobile agents

Encryption:

Encryption is powerful for providing privacy, authenticity, integrity, and limited access to data. Because networks often involve even greater risks, they often secure data with encryption, perhaps in combination with other controls. There are 2 types of encryption scheme exists:

- Link encryption (data are encrypted just before the system places them on the physical communications link)
- End-to-end encryption (provides security from one end of a transmission to the other)

Content Integrity:

Content integrity comes as a bonus with cryptography. No one can change encrypted data in a meaningful way without breaking the encryption. This does not say, however, that encrypted data cannot be modified. Changing even one bit of an encrypted data stream affects the result after decryption, often in a way that seriously alters the resulting plaintext. We need to consider three potential threats:

□ Malicious modification that changes content in a meaningful way

□ Malicious or non-malicious modification that changes content in a way that is not necessarily meaningful

□ non-malicious modification that changes content in a way that will not be detected

Encryption addresses the first of these threats very effectively. To address the others, we can use other controls.

Strong Authentication:

In the network case, however, authentication may be more difficult to achieve securely because of the possibility of eavesdropping and wiretapping, which are less common in non- networked environments. Also, both ends of a communication may need to be authenticated to each other.

Here the main issues are

- One time password
- Challenge response systems
- Digital distributed authentication

Access Controls:

Authentication deals with the *who* of security policy enforcement; access controls enforce the *what* and *how*.

ACLs on Routers

Routers perform the major task of directing network traffic either to sub-networks they control or to other routers for subsequent delivery to other sub-networks. Routers convert external IP addresses into internal MAC addresses of hosts on a local sub-network. Suppose a host is being spammed (flooded) with packets from a malicious rogue host. Routers can be configured with access control lists to deny access to particular hosts from particular hosts. So, a router could delete all packets with a source address of the rogue host and a destination address of the target host.

Alarms and Alerts:

The logical view of network protection looks like the figure below, in which both a router and a firewall provide layers of protection for the internal network. Now let us add one more layer to this defense.

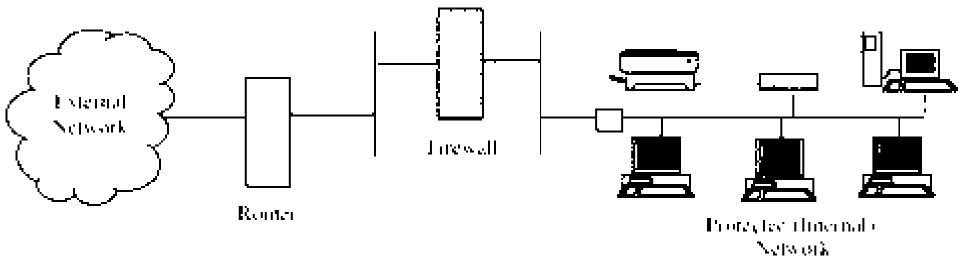


Fig. Layered network protection

Honey Pot: (A computer system open for attackers**)**

A honey pot has no special features. It is just a computer system or a network segment, loaded with servers and devices and data. It may be protected with a firewall, although you want the attackers to have some access. There may be some monitoring capability, done carefully so that the monitoring is not evident to the attacker.

We put up a honey pot for several reasons:

- To watch what attackers do, in order to learn about new attacks (so that you can strengthen your defenses against these new attacks)
- To lure an attacker to a place in which you may be able to learn enough to identify and stop the attacker
- To provide an attractive but diversionary playground, hoping that the attacker will leave your real system alone

Firewalls

Firewalls were officially invented in the early 1990s, but the concept really reflects the reference monitor from two decades earlier.

What is a Firewall?

A firewall is a device that filters all traffic between a protected or "inside" network and a less trustworthy or "outside" network. Usually, a firewall runs on a dedicated device; because it is a single point through which traffic is channeled, performance is important, which means non-firewall functions should not be done on the same machine. Because a firewall is executable code, an attacker could compromise that code and execute from the firewall's device. Thus, the fewer pieces of code on the device, the fewer tools the attacker would have by compromising the firewall. Firewall code usually runs on a proprietary or carefully minimized operating system. The purpose of a firewall is to keep "bad" things outside a protected environment. To accomplish that, firewalls implement a security policy that is specifically designed to address what bad things might happen. For example, the policy might be to prevent any access from outside (while still allowing traffic to pass *from* the inside *to* the outside). Alternatively, the policy might permit accesses only from certain places, from certain users, or for certain activities. Part of the challenge of protecting a network with a firewall is determining which security policy meets the needs of the installation.

Design of Firewalls:

A reference monitor must be

- Always invoked
- Tamperproof
- Small and simple enough for rigorous analysis

A firewall is a special form of reference monitor. By carefully positioning a firewall within a network, we can ensure that all network accesses that we want to control must pass through it. This restriction meets the "always invoked" condition. A firewall is typically well isolated, making it highly immune to modification. Usually, a firewall is implemented on a separate computer, with direct connections only to the outside and inside networks. This isolation is expected to meet the "tamperproof" requirement. And firewall designers strongly recommend keeping the functionality of the firewall simple.

Types of Firewalls:

Firewalls have a wide range of capabilities. Types of firewalls include

- Packet filtering gateways or screening routers
- Stateful inspection firewalls
- Application proxies
- Guards
- Personal firewalls

Packet Filtering Gateway:

A packet filtering gateway or screening router is the simplest, and in some situations, the most effective type of firewall. A packet filtering gateway controls access to packets on the basis of packet address (source or destination) or specific transport protocol type (such as HTTP web traffic). As described earlier in this chapter, putting ACLs on routers may severely impede their performance. But a separate firewall behind (on the local side) of the router can screen traffic before it gets to the protected network. Figure 7-34 shows a packet filter that blocks access from (or to) addresses in one network; the filter allows HTTP traffic but blocks traffic using the Telnet protocol.

Stateful Inspection Firewall:

Filtering firewalls work on packets one at a time, accepting or rejecting each packet and moving on to the next. They have no concept of "state" or "context" from one packet to the next. A stateful inspection firewall maintains state information from one packet to another in the input stream.

One classic approach used by attackers is to break an attack into multiple packets by forcing some packets to have very short lengths so that a firewall cannot detect the signature of an attack split across two or more packets. (Remember that with the TCP protocols, packets can arrive in any order, and the protocol suite is responsible for reassembling the packet stream in proper order before passing it along to the application.) A stateful inspection firewall would track the sequence of packets and conditions from one packet to another to thwart such an attack.

Application Proxy

Packet filters look only at the headers of packets, not at the data *inside* the packets. Therefore, a packet filter would pass anything to port 25, assuming its screening rules allow inbound connections to that port. But applications are complex and sometimes contain errors. Worse, applications (such as the e-mail delivery agent) often act on behalf of all users, so they require privileges of all users (for example, to store incoming mail messages so that inside users can read them). A flawed application, running with all users' privileges, can cause much damage. An application proxy gateway, also called a bastion host, is a firewall that simulates the (proper) effects of an application so that the application receives only requests to act properly. A proxy gateway is a two-headed device: It looks to the inside as if it is the outside (destination) connection, while to the outside it responds just as the insider would.

An application proxy runs pseudo-applications. For instance, when electronic mail is transferred to a location, a sending process at one site and a receiving process at the destination communicate by a protocol that establishes the legitimacy of a mail transfer and then actually transfers the mail message. The protocol between sender and destination is carefully defined. A proxy gateway essentially intrudes in the middle of this protocol exchange, seeming like a destination in communication with the sender that is outside the firewall, and seeming like the sender in communication with the real destination on the inside. The proxy in the middle has the opportunity to screen the mail transfer, ensuring that only acceptable e-mail protocol commands are sent to the destination.

Guard:

A guard is a sophisticated firewall. Like a proxy firewall, it receives protocol data units, interprets them, and passes through the same or different protocol data units that achieve either the same result or a modified result. The guard decides what services to perform on the user's behalf in accordance with its available knowledge, such as whatever it can reliably know of the (outside) user's identity, previous interactions, and so forth. The degree of control a guard can provide is limited only by what is computable. But guards and proxy firewalls are similar enough that the distinction between them is sometimes fuzzy. That is, we can add functionality to a proxy firewall until it starts to look a lot like a guard.

Personal Firewalls:

A personal firewall is an application program that runs on a workstation to block unwanted traffic, usually from the network. A personal firewall can complement the work of a conventional firewall by screening the kind of data a single host will accept, or it can compensate for the lack of a regular firewall, as in a private DSL or cable modem connection.

The personal firewall is configured to enforce some policy. For example, the user may decide that certain sites, such as computers on the company network, are highly trustworthy, but most other sites are not. The user defines a policy permitting download of code, unrestricted data sharing, and management access from the corporate segment, but not from other sites. Personal firewalls can also generate logs of accesses, which can be useful to examine in case something harmful does slip through the firewall.

A personal firewall runs on the very computer it is trying to protect. Thus, a clever attacker is likely to attempt an undetected attack that would disable or reconfigure the firewall for the future. Still, especially for cable modem, DSL, and other "always on" connections, the static workstation is a visible and vulnerable target for an ever-present attack community. A personal firewall can provide reasonable protection to clients that are not behind a network firewall.

Comparison of Firewall types:

Packet Filtering	Stateful Inspection	Application Proxy	Guard	Personal firewall
Simple	More complex	Even complex	Most complex	Similar to packet filtering
Sees only addresses and service protocol type	Can see either addresses or data	Sees full data portion of packet	Sees full text of communication	Can see full data portion of packet
Auditing difficult	Auditing possible	Can audit activity	Can audit activity	Can and usually does audit activity
Screens based on connection rules	Screens based on information across packets in either header or data field	Screens based on behavior of proxies	Screens based on interpretation of message contents	Typically, screens based on information in a single packet, using header or data
Complex addressing rules can make configuration tricky	Usually preconfigured to detect certain attack signatures	Simple proxies can substitute for complex addressing rules	Complex guard functionality can limit assurance	Usually starts in "deny all inbound" mode, to which user adds trusted addresses as they appear

Intrusion Detection System:

An intrusion detection system (IDS) is a device, typically another separate computer, that monitors activity to identify malicious or suspicious events. An IDS is a sensor, like a smoke detector, that raises an alarm if specific things occur. A model of an IDS is shown in below figure. The components in the figure are the four basic elements of an intrusion detection system, based on the Common Intrusion Detection Framework of [STA96]. An IDS receives raw inputs from sensors. It saves those inputs, analyzes them, and takes some controlling action.

Types of IDSs

The two general types of intrusion detection systems are signature based and heuristic. Signature-based intrusion detection systems perform simple pattern-matching and report situations that match a pattern corresponding to a known attack type. Heuristic intrusion detection systems, also known as anomaly based, build a model of acceptable behavior and flag exceptions to that model; for the future, the administrator can mark a flagged behavior as acceptable so that the heuristic IDS will now treat that previously unclassified behavior as acceptable.

Intrusion detection devices can be network based or host based. A network-based IDS is a stand-alone device attached to the network to monitor traffic throughout that network; a host-based IDS runs on a single workstation or client or host, to protect that one host.

Signature-Based Intrusion Detection:

A simple signature for a known attack type might describe a series of TCP SYN packets sent to many different ports in succession and at times close to one another, as would be the case for a port scan. An intrusion detection system would probably find nothing unusual in the first SYN, say, to port 80, and then another (from the same source address) to port 25. But as more and more ports receive SYN packets, especially ports that are not open, this pattern reflects a possible port scan. Similarly, some implementations of the protocol stack fail if they receive an ICMP packet with a data length of 65535 bytes, so such a packet would be a pattern for which to watch.

Heuristic Intrusion Detection:

Because signatures are limited to specific, known attack patterns, another form of intrusion detection becomes useful. Instead of looking for matches, heuristic intrusion detection looks for behavior that is out of the ordinary. The original work in this area focused on the individual, trying to find characteristics of that person that might be helpful in understanding normal and abnormal behavior. For example, one user might always start the day by reading e-mail, write many documents using a word processor, and occasionally back up files. These actions would be normal. This user does not seem to use many administrator utilities. If that person tried to access sensitive system management utilities, this new behavior might be a clue that someone else was acting under the user's identity.

Inference engines work in two ways. Some, called state-based intrusion detection systems, see the system going through changes of overall state or configuration. They try to detect when the system has veered into unsafe modes. Others try to map current activity onto a model of unacceptable activity and raise an alarm when the activity resembles the model. These are called model-based intrusion detection systems. This approach has been extended to networks in [MUK94]. Later work sought to build a dynamic model of behavior, to accommodate variation and evolution in a person's actions over time. The technique compares real activity with a known representation of normality.

Alternatively, intrusion detection can work from a model of known bad activity. For example, except for a few utilities (login, change password, create user), any other attempt to access a password file is suspect. This form of intrusion detection is known as misuse intrusion detection. In this work, the real activity is compared against a known suspicious area.

Stealth Mode:

An IDS is a network device (or, in the case of a host-based IDS, a program running on a network device). Any network device is potentially vulnerable to network attacks. How useful would an IDS be if it itself were deluged with a denial-of-service attack? If an attacker succeeded in logging in to a system within the protected network, wouldn't trying to disable the IDS be the next step?

To counter those problems, most IDSs run in stealth mode, whereby an IDS has two network interfaces: one for the network (or network segment) being monitored and the other to generate alerts and perhaps other administrative needs. The IDS uses the monitored interface as input only; it *never* sends packets out through that interface. Often, the interface is configured so that the device has no published address through the monitored interface; that is, a router cannot route anything to that address directly, because the router does not know such a device exists. It is the perfect passive wiretap. If the IDS needs to generate an alert, it uses only the alarm interface on a completely separate control network.

Goals for Intrusion Detection Systems:

1. *Responding to alarms:*

Whatever the type, an intrusion detection system raises an alarm when it finds a match. The alarm can range from something modest, such as writing a note in an audit log, to something significant, such as paging the system security administrator. Particular implementations allow the user to determine what action the system should take on what events. In general, responses fall into three major categories (any or all of which can be used in a single response):

- Monitor, collect data, perhaps increase amount of data collected
- Protect, act to reduce exposure
- Call a human

2. *False Results:*

Intrusion detection systems are not perfect, and mistakes are their biggest problem. Although an IDS might detect an intruder correctly most of the time, it may stumble in two different ways: by raising an alarm for something that is not really an attack (called a false positive, or type I error in the statistical community) or not raising an alarm for a real attack (a false negative, or type II error). Too many false positives mean the administrator will be less confident of the IDS's warnings, perhaps leading to a real alarm's being ignored. But false negatives mean that real attacks are passing the IDS without action. We say that the degree of false positives and false negatives represents the sensitivity of the system. Most IDS implementations allow the administrator to tune the system's sensitivity, to strike an acceptable balance between false positives and negatives.

IDS strength and limitations:

On the upside, IDSs detect an ever-growing number of serious problems. And as we learn more about problems, we can add their signatures to the IDS model. Thus, over time, IDSs continue to improve. At the same time, they are becoming cheaper and easier to administer. On the downside, avoiding an IDS is a first priority for successful attackers. An IDS that is not well defended is useless. Fortunately, stealth mode IDSs are difficult even to find on an internal network, let alone to compromise. IDSs look for known weaknesses, whether through patterns of known attacks or models of normal behavior. Similar IDSs may have identical vulnerabilities, and their selection criteria may miss similar attacks. Knowing how to evade a particular model of IDS is an important piece of intelligence passed within the attacker community. Of course, once manufacturers become aware of a shortcoming in their products, they try to fix it. Fortunately, commercial IDSs are pretty good at identifying attacks. Another IDS limitation is its sensitivity, which is difficult to measure and adjust. IDSs will never be perfect, so finding the proper balance is critical.

In general, IDSs are excellent additions to a network's security. Firewalls block traffic to particular ports or addresses; they also constrain certain protocols to limit their impact. But by definition, firewalls have to allow some traffic to enter a protected area.

Watching what that traffic actually does inside the protected area is an IDS's job, which it does quite well.

Secure Email:

We rely on e-mail's confidentiality and integrity for sensitive and important communications, even though ordinary e-mail has almost no confidentiality or integrity. Here we investigate how to add confidentiality and integrity protection to ordinary e-mail.

Security of email:

Sometimes we would like e-mail to be more secure. To define and implement a more secure form, we begin by examining the exposures of ordinary e-mail.

Threats to E-mail

- Message interception (confidentiality)
- Message interception (blocked delivery)
- Message interception and subsequent replay
- Message content modification
- Message origin modification
- Message content forgery by outsider
- Message origin forgery by outsider
- Message content forgery by recipient
- Message origin forgery by recipient
- Denial of message transmission

Requirements and solutions:

Following protections must be taken for protection in emails

- *Message confidentiality* (the message is not exposed end route to the receiver)
- *Message integrity* (what the receiver sees is what was sent)
- *Sender authenticity* (the receiver is confident who the sender was)
- *Non repudiation* (the sender cannot deny having sent the message)

Designs:

One of the design goals for encrypted e-mail was allowing security-enhanced messages to travel as ordinary messages through the existing Internet e-mail system. This requirement ensures that the large existing e-mail network would not require change to accommodate security. Thus, all protection occurs within the body of a message.

Confidentiality:

The encrypted e-mail standard works most easily as just described, using both symmetric and asymmetric encryption. The standard is also defined for symmetric encryption only: To use symmetric encryption, the sender and receiver must have previously established a shared secret encryption key. The processing type ("Proc-Type") field tells what privacy enhancement services have been applied. In the data exchange key field ("DEK-Info"), the kind of key exchange (symmetric or asymmetric) is shown. The key exchange ("Key-Info") field contains the message encryption key, encrypted under this shared encryption key. The field also identifies the originator (sender) so that the receiver can determine which shared symmetric key was used. If the key exchange technique were to use asymmetric encryption, the key exchange field would contain the message encryption field, encrypted under the recipient's public key. Also included could be the sender's certificate (used for determining authenticity and for generating replies). The encrypted e-mail standard supports multiple encryption algorithms, using popular algorithms such as DES, triple DES, and AES for message confidentiality, and RSA and Diffie-Hellman for key exchange.

Encryption of secure e-mail:

Encrypted e-mail provides strong end-to-end security for electronic mail. Triple DES, AES, and RSA cryptography are quite strong, especially if RSA is used with a long bit key (1024 bits or more). The vulnerabilities remaining with encrypted e-mail come from the points not covered: the endpoints. An attacker with access could subvert a sender's or receiver's machine, modifying the code that does the privacy enhancements or arranging to leak a cryptographic key.

Examples of Secure E-mail:

- PGP (Pretty Good Privacy)
- S/MIME (Secure Multipurpose Internet Mail Extensions)

<u>EXERCISES</u>

1. The FTP protocol is relatively easy to proxy; the firewall decides, for example, whether an outsider should be able to access a particular directory in the file system and issues a corresponding command to the inside file manager or responds negatively to the outsider. Other protocols are not feasible to proxy.
   List three protocols that it would be prohibitively difficult or impossible to proxy. Explain your answer.
2. How would the content of the audit log differ for a screening router versus an application proxy firewall?
3. Cite a reason why an organization might want two or more firewalls on a single network.
4. Firewalls are targets for penetrators. Why are there few compromises of firewalls?

5. Should a network administrator put a firewall in front of a honey pot? Why or why not?
6. Can a firewall block attacks using server scripts, such as the attack in which the user could change a price on an item offered by an e-commerce site? Why or why not?
7. Why does a stealth mode IDS need a separate network to communicate alarms and to accept management commands?
8. One form of IDS starts operation by generating an alert for every action. Over time, the administrator adjusts the setting of the IDS so that common, benign activities do not generate alarms. What are the advantages and disadvantages of this design for an IDS?
9. Can encrypted e-mail provide verification to a sender that a recipient has read an e-mail message? Why or why not?
10. Can message confidentiality and message integrity protection be applied to the same message? Why or why not?
11. What are the advantages and disadvantages of an e-mail program that automatically applies and removes protection to e-mail messages between sender and receiver?

MODULE 3

Administering Security

Security planning:

Contents of security planning:

A security plan identifies and organizes the security activities for a computing system. The plan is both a description of the current situation and a plan for improvement. Every security plan must address seven issues.
1. P*olicy*, indicating the goals of a computer security effort and the willingness of the people involved to work to achieve those goals
2. *Current state*, describing the status of security at the time of the plan
3. Requirements, recommending ways to meet the security goals
4. R*ecommended controls*, mapping controls to the vulnerabilities identified in the policy and requirements
5. *Accountability*, describing who is responsible for each security activity
6. *Timetable*, identifying when different security functions are to be done
7. *Continuing attention*, specifying a structure for periodically updating the security plan

1. Policy:

The policy statement should specify the following:

- The organization's *goals* on security. For example, should the system protect data from leakage to outsiders, protect against loss of data due to physical disaster, protect the data's integrity, or protect against loss of business when computing resources fail?

What is the higher priority: serving customers or securing data?

- Where the *responsibility* for security lies. For example, should the responsibility rest with a small computer security group, with each employee, or with relevant managers?
- The organization's *commitment* to security. For example, who provides security support for staff, and where does security fit into the organization's structure?

2. Current Security Status:

To be able to plan for security, an organization must understand the vulnerabilities to which it may be exposed. The organization can determine the vulnerabilities by performing a risk analysis: a careful investigation of the system, its environment, and the things that might go wrong. The risk analysis forms the basis for describing the current status of security. The status can be expressed as a listing of organizational assets, the security threats to the assets, and the controls in place to protect the assets.

The status portion of the plan also defines the limits of responsibility for security. It describes not only which assets are to be protected but also who is responsible for protecting them. The plan may note that some groups may be excluded from responsibility; for example,

joint ventures with other organizations may designate one organization to provide security for all
member organizations. The plan also defines the boundaries of responsibility, especially when networks are involved. For instance, the plan should clarify who provides the security for a network router or for a leased line to a remote site.
Even though the security plan should be thorough, there will necessarily be vulnerabilities that are not considered. These vulnerabilities are not always the result of ignorance rather, they can arise from the addition of new equipment or data as the system evolves.
They can also result from new situations, such as when a system is used in ways not anticipated by its designers. The security plan should detail the process to be followed when someone identifies a new vulnerability. In particular, instructions should explain how to integrate controls for that vulnerability into the existing security procedures.

3. Requirements:

The heart of the security plan is its set of security requirements: functional or performance demands placed on a system to ensure a desired level of security. The requirements are usually derived from organizational needs. Sometimes these needs include the need to conform to specific security requirements imposed from outside, such as by a government agency or a commercial standard.

4. Recommended Controls:

The security requirements lay out the system's needs in terms of what should be protected. The security plan must also recommend what controls should be incorporated into the system to meet those requirements. Throughout this book you have seen many examples of controls, so we need not review them here. As we see later in this chapter, we can use risk analysis to create a map from vulnerabilities to controls. The mapping tells us how the system will meet the security requirements. That is, the recommended controls address implementation issues: how the system will be designed and developed to meet stated security requirements.

5. Responsibility for Implementation:

A section of the security plan should identify which people are responsible for implementing the security requirements. This documentation assists those who must coordinate their individual responsibilities with those of other developers. At the same time, the plan makes explicit who is accountable should some requirement not be met or some vulnerability not be addressed. That is, the plan notes who is responsible for implementing controls when a new vulnerability is discovered or a new kind of asset is introduced.

People building, using, and maintaining the system play many roles. Each role can take some responsibility for one or more aspects of security. Consider, for example, the groups listed here.

 Personal computer users may be responsible for the security of their own machines. Alternatively, the security plan may designate one person or group to be coordinator of personal computer security.
 Project leaders may be responsible for the security of data and computations.

6. Timetable:

A comprehensive security plan cannot be executed instantly. The security plan includes a timetable that shows how and when the elements of the plan will be performed. These dates also give milestones so that management can track the progress of implementation.

7. Continuing Attention:

Good intentions are not enough when it comes to security. We must not only take care in defining requirements and controls, but we must also find ways for evaluating a system's security to be sure that the system is as secure as we intend it to be. Thus, the security plan must call for reviewing the security situation periodically. As users, data, and equipment change, new exposures may develop. In addition, the current means of control may become obsolete or ineffective (such as when faster processor times enable attackers to break an encryption algorithm). The inventory of objects and the list of controls should periodically be scrutinized and updated, and risk analysis performed anew.

Security Planning Team Members:

The membership of a computer security planning team must somehow relate to the different aspects of computer security described in this book. Security in operating systems and networks requires the cooperation of the systems administration staff. Program security measures can be understood and recommended by applications programmers. Physical security controls are implemented by those responsible for general physical security, both against human attacks and natural disasters. Finally, because controls affect system users, the plan should incorporate users' views, especially with regard to usability and the general desirability of controls.
Thus, no matter how it is organized, a security planning team should represent each of the following groups.

☐ Computer hardware group
☐ System administrators
☐ Systems programmers
☐ Applications programmers
☐ Data entry personnel
☐ Physical security personnel
☐ Representative users

In some cases, a group can be adequately represented by someone who is consulted at appropriate times, rather than a committee member from each possible constituency being enlisted.

Assuring Commitment To a security plan:

After the plan is written, it must be accepted and its recommendations carried out. Acceptance by the organization is key; a plan that has no organizational commitment is simply a plan that collects dust on the shelf. Commitment to the plan means that security functions will be implemented and security activities carried out. Three groups of people must contribute to making the plan a success.

☐ The planning team must be sensitive to the needs of each group affected by the plan.

 Those affected by the security recommendations must understand what the plan means for the way they will use the system and perform their business activities. In particular, they must see how what they do can affect other users and other systems.
 Management must be committed to using and enforcing the security aspects of the system.

Management commitment is obtained through understanding. But this understanding is not just a function of what makes sense technologically; it also involves knowing the cause and the potential effects of lack of security. Managers must also weigh tradeoffs in terms of convenience and cost. The plan must present a picture of how cost effective the controls are, especially when compared to potential losses if security is breached without the controls. Thus, proper presentation of the plan is essential, in terms that relate to management as well as technical concerns.
Management is often reticent to allocate funds for controls until the value of those controls is explained. As we note in the next section, the results of a risk analysis can help communicate the financial tradeoffs and benefits of implementing controls. By describing vulnerabilities in financial terms and in the context of ordinary business activities (such as leaking data to a competitor or an outsider), security planners can help managers understand the need for controls.
The plans we have just discussed are part of normal business. They address how a business handles computer security need. Similar plans might address how to increase sales or improve product quality, so these planning activities should be a natural part of management. Next, we turn to two particular kinds of business plans that address specific security problems: coping with and controlling activity during security incidents.

Business Continuity Plan:

A business continuity plan documents how a business will continue to function during a computer security incident. An ordinary security plan covers computer security during normal times and deals with protecting against a wide range of vulnerabilities from the usual sources.

A business continuity plan deals with situations having two characteristics:

- C*atastrophic situations*, in which all or a major part of a computing capability is suddenly unavailable
- L*ong duration*, in which the outage is expected to last for so long that business will suffer

There are many situations in which a business continuity plan would be helpful. Here are some examples that typify what you might find in reading your daily newspaper:

- A fire destroys a company's entire network.
- A seemingly permanent failure of a critical software component renders the computing system unusable.
- A business must deal with the abrupt failure of its supplier of electricity, telecommunications, network access, or other critical service.
- A flood prevents the essential network support staff from getting to the operations center.

The key to coping with such disasters is advance planning and preparation, identifying activities that will keep a business viable when the computing technology is disabled. The steps in business continuity planning are these:

- Assess the business impact of a crisis.
- Develop a strategy to control impact.

Develop and implement a plan for the strategy

Incident response plan:

Incident response Plan should be

- define what constitutes an *incident*
- identify who is responsible for *taking charge* of the situation
- describe the plan of *action*

Risk Analysis:

We distinguish a risk from other project events by looking for three things,
1. *A loss associated with an event*. The event must generate a negative effect: compromised security, lost time, diminished quality, lost money, lost control, lost understanding, and so on. This loss is called the risk impact.
2. *The likelihood that the event will occur*. The probability of occurrence associated with each risk is measured from 0 (impossible) to 1 (certain). When the risk probability is 1, we say we have a problem.
3. *The degree to which we can change the outcome*. We must determine what, if anything, we can do to avoid the impact or at least reduce its effects. Risk control involves a set of actions to reduce or eliminate the risk.

We usually want to weigh the pros and cons of different actions we can take to address each risk. To that end, we can quantify the effects of a risk by multiplying the risk impact by the risk probability, yielding the risk exposure. For example, if the likelihood of virus attack is 0.3 and the cost to clean up the affected files is $10,000, then the risk exposure is $3,000. So, we can use a calculation like this one to decide that a virus checker is worth an investment of $100, since it will prevent a much larger potential loss. Clearly, risk probabilities can change over time, so it is important to track them and plan for events accordingly.

Risk is inevitable in life: Crossing the street is risky but that does not keep us from doing it. We can identify, limit, avoid, or transfer risk but we can seldom eliminate it. In general, we have three strategies for dealing with risk:
1. *Avoiding* the risk, by changing requirements for security or other system characteristics
2. *Transferring* the risk, by allocating the risk to other systems, people, organizations, or assets; or by buying insurance to cover any financial loss should the risk become a reality
3. *Assuming* the risk, by accepting it, controlling it with available resources, and preparing to deal with the loss if it occurs
Thus, costs are associated not only with the risk's potential impact but also with reducing it. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. In other words, risk leverage is

$$\frac{(\text{risk exposure before reduction}) - (\text{risk exposure after reduction})}{(\text{cost of risk reduction})}$$

The Nature of Risk:

In our everyday lives, we take risks. In crossing the road, eating oysters, or playing the lottery, we take the chance that our actions may result in some negative result such as being injured, getting sick, or losing money. Consciously or unconsciously, we weigh the benefits of taking the action with the possible losses that might result. Just because there is a risk to a certain act, we do not necessarily avoid it; we may look both ways before crossing the street, but we do cross it. In building and using computing systems, we must take a more organized and careful approach to assessing our risks. Many of the systems we build and use can have a dramatic impact on life and health if they fail. For this reason, risk analysis is an essential part of security planning.

We cannot guarantee that our systems will be risk free; that is why our security plans must address actions needed should an unexpected risk become a problem. And some risks are simply part of doing business; for example, as we have seen, we must plan for disaster recovery, even though we take many steps to avoid disasters in the first place.

When we acknowledge that a significant problem cannot be prevented, we can use controls to reduce the seriousness of a threat. For example, you can back up files on your computer as a defense against the possible failure of a file storage device. But as our computing systems become more complex and more distributed, complete risk analysis becomes more difficult and time consuming and more essential.

Steps of a Risk Analysis:

Risk analysis is performed in many different contexts; for example, environmental and health risks are analyzed for activities such as building dams, disposing of nuclear waste, or changing a manufacturing process. Risk analysis for security is adapted from more general management practices, placing special emphasis on the kinds of problems likely to arise from security issues. By following well-defined steps, we can analyze the security risks in a computing system.

The basic steps of risk analysis are listed below.

1. Identify assets.
2. Determine vulnerabilities.
3. Estimate likelihood of exploitation.
4. Compute expected annual loss.
5. Survey applicable controls and their costs.
6. Project annual savings of control.

Arguments For and against risk analysis:

Risk analysis is a well-known planning tool, used often by auditors, accountants, and managers. In many situations, such as obtaining approval for new drugs, new power plants, and new medical devices, a risk analysis is required by law in many countries. There are many good reasons to perform a risk analysis in preparation for creating a security plan.

- *Improve awareness.* Discussing issues of security can raise the general level of interest and concern among developers and users. Especially when the user population has little expertise in computing, the risk analysis can educate users about the role security plays in protecting functions and data that are essential to user operations and products.
- *Relate security mission to management objectives.* Security is often perceived as a financial drain for no gain. Management does not always see that security helps balance harm and control costs.
- *Identify assets, vulnerabilities, and controls.* Some organizations are unaware of their computing assets, their value to the organization, and the vulnerabilities associated

with those assets. A systematic analysis produces a comprehensive list of assets, valuations, and risks.

- *Improve basis for decisions.* A security manager can present an argument such as "I think we need a firewall here" or "I think we should use token-based authentication instead of passwords." Risk analysis augments the manager's judgment as a basis for the decision. *Justify expenditures for security.* Some security mechanisms appear to be very expensive and without obvious benefit. A risk analysis can help identify instances where it is worth the expense to implement a major security mechanism. Justification is often derived from examining the much larger risks of *not* spending for security.

Organizational Security Policies:

A security policy is a high-level management document to inform all users of the goals of and constraints on using a system. A policy document is written in broad enough terms that it does not change frequently. The information security policy is the foundation upon which all protection efforts are built. It should be a visible representation of priorities of the entire organization, definitively stating underlying assumptions that drive security activities. The policy should articulate senior management's decisions regarding security as well as asserting management's commitment to security. To be effective, the policy must be understood by everyone as the product of a directive from an authoritative and influential person at the top of the organization.

Purpose:

Security policies are used for several purposes, including the following:

- recognizing sensitive information assets
- clarifying security responsibilities
- promoting awareness for existing employees
- guiding new employees

Audience:

A security policy addresses several different audiences with different expectations. That is, each group users, owners, and beneficiaries use the security policy in important but different ways.

Users

Users legitimately expect a certain degree of confidentiality, integrity, and continuous availability in the computing resources provided to them. Although the degree varies with the situation, a security policy should reaffirm a commitment to this requirement for service.
Users also need to know and appreciate what is considered acceptable use of their computers, data, and programs. For users, a security policy should define acceptable use.

Owners

Each piece of computing equipment is owned by someone, and the owner may not be a system user. An owner provides the equipment to users for a purpose, such as to further education, support commerce, or enhance productivity. A security policy should also reflect the expectations and needs of owners.

Beneficiaries

A business has paying customers or clients; they are beneficiaries of the products and services offered by that business. At the same time, the general public may benefit in several ways: as a source of employment or by provision of infrastructure.

Contents:

A security policy must identify its audiences: the beneficiaries, users, and owners. The policy should describe the nature of each audience and their security goals. Several other sections are required, including the purpose of the computing system, the resources needing protection, and the nature of the protection to be supplied.

- Purpose
- Protected resources
- Nature of protection

Characteristics of a Good Security Policy:

If a security policy is written poorly, it cannot guide the developers and users in providing appropriate security mechanisms to protect important assets. Certain characteristics make a security policy a good one.

- Durability
- Realism
- Usefulness

Physical security

Physical security is the term used to describe protection needed outside the computer system. Typical physical security controls include guards, locks, and fences to deter direct attacks. In addition, there are other kinds of protection against less direct disasters, such as floods and power outages; these, too, are part of physical security.

Natural Disasters:

It is impossible to prevent natural disasters, but through careful planning it is possible to reduce the damage they inflict. Some measures can be taken to reduce their impact. Because many of these perils cannot be prevented or predicted, controls focus on limiting possible damage and recovering quickly from a disaster. Issues to be considered include the need for offsite backups, the cost of replacing equipment, the speed with which equipment can be replaced, the need for available computing power, and the cost or difficulty of replacing data and programs. Some of them are

- Flood
- Fire
- Other natural disasters

Power loss:

Computers need their food electricity and they require a constant, pure supply of it. With a direct power loss, all computation ceases immediately. Because of possible damage to media by sudden loss of power, many disk drives monitor the power level and quickly retract the recording head if power fails. For certain time-critical applications, loss of service from the system is intolerable; in these cases, alternative complete power supplies must be instantly available.

Human vandals:

Because computers and their media are sensitive to a variety of disruptions, a vandal can destroy hardware, software, and data. Human attackers may be disgruntled employees, bored operators, saboteurs, people seeking excitement, or unwitting bumblers. If physical access is easy to obtain, crude attacks using axes or bricks can be very effective. One man recently shot a computer that he claimed had been in the shop for repairs many times without success. Physical attacks by unskilled vandals are often easy to prevent; a guard can stop someone approaching a computer installation with a threatening or dangerous object. When physical access is difficult, more subtle attacks can be tried, resulting in quite serious damage. People with only some sophisticated knowledge of a system can short-circuit a computer with a car key or disable a disk drive with a paper clip. These items are not likely to attract attention until the attack is completed.

- Unauthorized access and use
- Theft
- Preventing access
- Preventing portability
- Detecting theft

Interception of Sensitive Information:

When disposing of a draft copy of a confidential report containing its sales strategies for the next five years, a company wants to be especially sure that the report is not reconstruct able by one of its competitors. When the report exists only as hard copy, destroying the report is straightforward, usually accomplished by shredding or burning. But when the report exists digitally, destruction is more problematic. There may be many copies of the report in digital and paper form and in many locations (including on the computer and on storage media).
There may also be copies in backups and archived in e-mail files. Here, we look at several ways to dispose of sensitive information. They are

- Shredding
- Overwriting magnetic data
- Degaussing
- Protecting against Emanation

Contingency Planning:

The key to successful recovery is adequate preparation. Seldom does a crisis destroy irreplaceable equipment; most computing systems personal computers to mainframes are standard, off-the-shelf systems that can be easily replaced. Data and locally developed programs are more vulnerable because they cannot be quickly substituted from another source. Let us look what to do after a crisis occurs.

- Back-up
- Off-site backup
- Network storage
- Cold site
- Hot site

Physical security backup:

We have to protect the facility against many sorts of disasters, from weather to chemical spills and vehicle crashes to explosions. It is impossible to predict what will occur or when. The physical security manager has to consider all assets and a wide range of harm. Malicious humans seeking physical access are a different category of threat agent. The primary physical controls are strength and duplication. Strength means overlapping controls implementing a defense-in-depth approach so that if one control fails, the next one will protect. People who built ancient castles practiced this philosophy with moats, walls, drawbridges, and arrow slits. Duplication means eliminating single points of failure. Redundant copies of data protect against harm to one copy from any cause. Spare hardware components protect against failures.

EXERCISES

1. In what ways is denial of service (lack of availability for authorized users) a vulnerability to users of single-user personal computers?
2. List three factors that should be considered when developing a security plan.
3. Cite three controls that could have both positive and negative effects.
4. List three different sources of water to a computing system, and state a control for each.
5. Cite a risk in computing for which it is impossible or infeasible to develop a classical probability of occurrence.
6. Investigate the computer security policy for your university or employer. Who wrote the policy? Who enforces the policy? Who does it cover? What resources does it cover?
7. For an airline, what are its most important assets? What are the minimal computing resources it would need to continue business for a limited period (up to two days)? What other systems or processes could it use during the period of the disaster?
8. Investigate your university's or employer's security plan to determine whether its security requirements meet all the conditions listed in this chapter. List any that do not. When was the plan written? When was it last reviewed and updated?

LEGAL, PRIVACY, AND ETHIAL ISSUES IN COMPUTER SECURITY

In this Chapter

- Program and data protection by patents, copyrights, and trademarks
- Computer crime
- Privacy
- Ethical analysis of computer security situations

Protecting Programs and Data

Copyrights, patents, and trade secrets are legal devices that can protect computers, programs and data. Here how each of these forms are originally designed to be used and how each is currently used in computing are described.

Copyrights: Copyrights are designed to protect the expression of ideas. Thus, it is applicable to a creative work, such as story, photographs, song or pencil sketch. The right to copy an expression of an idea is protected by copyright. The idea of copyright is to allow regular and free exchange of ideas. Copyright gives the author the exclusive right to make copies of the expression and sell them in public. That is, only the author can sell the copies of the author's book.

Patents:

Patents are unlike copyrights in that they protect inventions, tangible objects, or ways to make them, not works of the mind. The distinction between patents and copyrights is that patents were intended to apply to the results of science, technology, and engineering, whereas copyrights are meant to cover works in the arts, literature, and written in the scholarship. A Patent is designed to protect the device or process for carrying out an idea itself.

Trade Secrets:

A trade secret is unlike a patent and copyright in that it must kept *secret*. The information has value only as secret, and an infringer is one who divulges the secret. Once divulged, the information usually cannot be made secret. A trade secret is information that gives one company a competitive edge over others. For example, the formula of a soft drink is a trade secret, as is a mailing list of customer or information about a product due to be announced in a few months.

Computer Crime:

Crimes involving computers are an area of the law that is even less clear than the other areas. Computer crime consider why new laws are needed to address some of its problems.

Issues in computer crime are

- Rules of property
- Rules of evidence
- Threats to integrity and confidentiality
- Value of data
- Acceptance of computer terminology

Why Computer crime is hard to define?

Some people in the legal process do not understand computers and computing, so crimes involving computers are not always treated properly. Main reasons are

1. Lack of understanding
2. Lack of physical evidence
3. Lack of recognition assets
4. Lack of political impacts
5. Complexity of case
6. Juveniles

Privacy:

In particular, we want to investigate the privacy of sensitive data about the user. The user should be protected against the system's misuse of the private data and the system's failure to protect its user's private data against outside attack and disclosure. This is termed as privacy in computer ethics.

Ethical Issues in Computer Security:

The primary purpose of this section is to explore some of ethical issues associated with computer security and to show how ethics functions as a control.

Difference between Law and Ethics:

Law	**Ethics**
Described by formal, written documents	Described by unwritten principles
Interpreted by courts	Interpreted by each individual
Established by legislature representing all people	Presented by philosophers, religions, professional groups
Applicable to everyone	Personal choice
Priority determined by courts if two laws conflict	Priority determined by an individual if two principles conflict
Court final arbiter of "right"	No external arbiter

Enforceable by police and court	Limited enforcement

Studying Ethics:

The study of ethics is not so easy because the issues are complex. Sometimes people confuse between ethics and religion because many religions provide a framework in which to make ethical choices. Here some of the problems and how understanding of ethics can deal with issues of computer security is explained.

- Ethics and religion
- Ethical principles are not universal\Ethics does not provide answers

Solutions to the issues:

1. Ethical reasoning
2. Examining the case for ethical issues
   Here some steps are used to make ethical choices justifiable. Those are
   I. Understanding the situation
   II. Know several theories of ethical reasoning
   III. List the ethical principles involved
   IV. Determine which principles outweigh others

Examples of ethical principle:

1. Consequence based principles
2. Rule based principles

Taxonomy o Ethical theories:

	Consequence based	Rule based
Individual	Based on consequences to individual	Based on rules acquired by the individual from religion experience, analysis
Universal	Based on consequences to all of society	Based on universal rules, evident to everyone

REFERENCES

1. Security in Computing – (3rd Edition) Charles P.Pfleeger, Shari Lawrence Pfleeger. PHI.

2. Cryptography and Network Security – by A. Kahate – TMH.