CHAPTER

13 Protection and Security

Learning Objectives

After reading this chapter, you should be able to:

- Explain the need for protection and security in operating systems.
- Distinguish between protection and security.
- Differentiate between software security and application security.
- Describe security vulnerabilities and types of security attacks.
- Differentiate between authorization and authentication.
- Explain access control-based security system.
- Describe protection matrix, capability list, and access control list.
- Describe viruses and worms.

13.1 Introduction

As microprocessor technology advances rapidly, computing is becoming increasingly ubiquitous and pervasive. With the surge in the Internet and communication technologies, the world is becoming highly interconnected. Organizations such as financial institutions, government offices, the commercial sector, health and educational institutions, military organizations, etc., are becoming increasingly integrated or interdependent by using computing and communication technologies. In a world that is highly dynamic and closely interconnected protection and security of computer systems are becoming vital requisites. Computer security is reaching global proportions with new security threats appearing by the day on the Internet. The penalty for slackness in security could range from a simple nuisance to a serious financial loss or national security breach. As a result, the task of providing security for computer users is proving to be a constant challenge. The

operating system is primarily responsible for serving computer users locally and remotely through the Internet and plays a major role in providing security necessary to protect user data and system resources.

The terms protection and security are employed with different connotations but, sometimes, they mean the same. For example, in one interpretation, protection means internally guarding resources and processes within a computer system whereas security is guarding the computer system from the external environment in which the computer resides. Another interpretation is that protection is to prevent unauthorized access, attack, or invasion, whereas security is considered as a means of ensuring a safe and trusted environment. We prefer the latter interpretation and use it as an informal definition of protection and security in this book. That is, security is a set of policies to ensure a safe and trusted environment for computer users, and protection is a collection of mechanisms to implement the security policies in the system. In essence, security is the goal and protection is the means to attain that goal.

In a broader sense, with respect to operating systems, protection may be viewed as its internal requirement. Security, on the other hand, deals with protecting a computer system from threats arising from the external environment in which the computer resides. Where the latter is concerned, the operating system may need to do much more than merely helping users run their applications. Primarily, the system must have a mechanism to identify users who are eligible to use the computer. The system employs security measures to insure that only eligible users have access to the resources. It must have a means to authenticate eligible users who attempt to use the system. The operative word here is authentication.

As we mentioned earlier, in the security domain, *protection* is a collection of mechanisms to implement the security policies of the operating system. Protection mechanisms regulate the execution of application programs, and the reading, modifying, and destruction of system data by them. The mechanisms must prevent accidental- or malicious disclosure, modification, or destruction of data.

Various protection mechanisms have been employed in modern operating systems. Some examples are (1) processor operating modes (e.g., user mode vs. kernel mode), (2) memory management registers (e.g., base and bound registers), (3) permission information in file metadata (e.g., read, write, execution bits), etc. Protection is a pervasive property, and is present in almost all components of the operating system. Protection mechanisms enhance the reliability and dependability of operating systems, and assist users in early detection of programming defects.

Each access from all application processes to every resource is checked against the authority the process owner has on the resource. The check is necessary to confine errors or infringements to the application processes themselves. An application process must not damage the operating system or other application processes. In Chapter 8, we investigated the roles of the memory management hardware and software in ensuring some level of protection in the system. Other components of the operating system employ

» Computer security has many aspects in common with human security. We as humans constantly deal with people and battle diseases that affect our wealth and health (mental and physical). As we enter the cyber world, a similar struggle can be expected to prevail for computer systems. Among the security issues of the physical world, some threats can be eliminated; some are chronic and, therefore, can only be controlled, while some others are difficult or impossible to deal with. This is true of the cyber world too. As the cyber world integrates with the physical world, security issues of future generations are only likely to increase and grow more complicated.

different mechanisms to ensure protection to the resources they manage. Designing suitable protection mechanisms and then implementing them is the way to assure security of hardware and software resources of a system.

To protect resources in the system, the operating system constructs tables of information on how they must be accessed (read, write, execute). Memory access is usually enforced by the hardware. In Chapter 8, we discussed the necessity of protecting the operating system from application processes, and of protecting one application process from others in multiprocess systems. The general protection problem in computer systems is much deeper and involved than that. It encompasses more than mere memory- and file protection.

Protection is a very general term we use for all the mechanisms the operating system employs to control accesses from the application processes to system resources. The operating system must make the computer a secure and reliable device even in the face of illegal users trying to break into the computer system. The security subsystem has to prevent illegal users from using the computer system and provide all legal users a reasonable assurance of security. In essence, protection refers to a collection of mechanisms to protect resources from illegitimate accesses from the processes.

In this chapter, we look at some of the issues related to security and protection in computer systems. Computer security and protection are very broad topics and, therefore, it is difficult to provide even a reasonable coverage of these topics in books on operating systems. Our main objective, therefore, is to provide only a broad overview of these important topics. In addition, the focus is on security and protection issues in operating systems.

Security violations may result in some form of damage to the system or its users. Similar damages occur due to accidents such as natural disasters (earthquakes, flood, fire, etc.), errors in hardware- and software components (program bugs, inaccessible devices, memory failure, etc.), and human errors (incorrect input, incorrect logic, mounting incorrect file system, etc.). These kinds of accidental damages are handled by storing necessary backups; this chapter does not deal with such accidental damages.

≫ *Access* to a system or its resources is the key term in security. Security is the ability to exercise control over a system (local or remote) or on a resource in a system. In this sense, security is establishing suitable control over the access to the system in general and resources in particular.

13.2 The Need for Security

In multiuser systems, several users store their data (and programs) in the system for long durations. The system data is also stored there, and is shared by all users. Data is stored in the abstractions of files. Because many users use a multiuser system, simultaneously or exclusively, there is a critical need to protect files from unauthorized use. For example, a database management system stores data in database files. These data constitute vital assets for the organizations owning them, and the consequences of their falling into wrong hands would be disastrous. Another example is of classified military data that needs to be protected from all unauthorized users. That is to say, such data must be protected from accidental- or intentional modifications, destruction, or disclosure, while at the same time other legitimate users must not suffer denial of use of their own data.

A system that allows many users to share its resources has to make prevention of abuse of resources its primary objective; this implies a means in the system to protect these shared resources lest they become unusable due to unpredictable abuse. The system has to ensure that users manipulate the resources in the manner prescribed by their owners or the system. The question is how the operating system protects resources (both hardware and software) against unauthorized accesses. The watchword here is unauthorized access.

Not all users (of a computer system) are allowed all manner of access to every resource in the system. For example, in the corporate world, employees do not have access to information relating to their peers' compensation, while their managers do. Similarly, not all computer users have identical authority to use resources of the system. Each user's usage is restricted to a subset of operations on a subset of resources. We need two things: (1) the means to inform the operating system about a user's authority for various resources; (2) the protection system must have the means to enforce the authority structure.

Protection is a way to control sharing of resources among multiple users. The protection subsystem must prevent- and trap unauthorized accesses to resources and so protection mechanisms of most operating systems are designed to confine errors of one user from damaging the system or other users. Most application programs tend to have bugs in their codes. Executions of such programs should not hinder other users from using the system. The operating system must check all accesses to every resource for the user's authority.

Apart from this, the operating system must ensure privacy of the data. It should adequately safeguard sensetive data by allowing access only to authorized users. Even if unauthorized users gain access to data, they must not be able to infer appropriate information from the data.

Security requirements vary from one system to another. Small computers, used by a closed group of few people, may not need the same level of security (and related protection) as those of large corporate systems. Such requirements are to be borne in mind in designing an operating system, and are preferable to redesigning them later to accommodate new specific security requirements. An operating system usually provides a basic set of security checking mechanisms that can adapt to meet most requirements.

In short, the security objectives of an operating system are: (1) to prevent illegal users from using the system, (2) to make unauthorized actions impossible, (3) to confine errors to the immediate contexts of their occurrences, and (4) to detect- and correct damage before it spreads to other parts of the system. The aim is to design- and develop systems that satisfy security requirements and while also meeting the compulsions of flexibility and resource-sharing necessary for market acceptability. These are indeed ambitious goals. In this chapter, we discuss security related issues from operating system perspectives.

≫ The operating system is the sole manager of a computer system. The protection problem drills down to restricting the operating system's activities when it works on behalf of application processes. It must not perform actions that the owners of those processes are not authorized to perform. The motto of security and protection is to restrain the operating system's activities when processes belonging to a particular user are active in the kernel space. Thereby, the operating system behaviour is dynamic, and context sensitive to different users.

13.3 Secure Software Systems

Building secure software systems is difficult, because computer security is not always a feature or a function of the system. It is often an emergent property or behaviour such as reliability and performance of the system. Emergent behaviours usually arise from the interaction of simple low-level components of the system. For example, a race condition is an emergent property and failure of the system due to race condition is considered undesirable security behaviour.

In complex dynamic systems, it is not humanly possible to visualize all potential emergent behaviours of a system before it is built and used. Therefore, avoiding all unwanted emergent behaviours from the system in advance is well nigh impossible. However, careful design of the operating system at the preliminary stage can avoid the most obvious and undesirable ones. The bottom-line is that software security cannot be added as a separate component of the system after it is built. Security must be dealt with throughout the developmental cycles. It must be designed and build into the system from ground up. Operating system is an important and main software component of a computer system and, therefore, incorporating proper security design practices into operating system design is extremely crucial for the security of any computer system.

The complexity of securing a computer system lies in the sophisticated protection mechanisms necessary for rigorous security and user convenience, but all have to be at a reasonable cost. However, some security schemes users may want are theoretically impossible or prohibitively expensive. Foolproof security is impossible, as no security assurance is perfect. However, some form of effective security is imperative. The goal is a cost-effective security system that reduces the risk of security breaches to a minimal level.

≫ It is difficult to define computer security precisely. Security problems differ from system to system, and the definition of security would accordingly differ. A system secured for one environment may be unsuitable for another, and vice versa. Operating system designers must define what 'security' means for their particular needs. For most systems, unauthorized disclosure, modification, or destruction of information are the most serious security threats. If there are security related weaknesses in the operating system, unscrupulous users may take undue advantage of those weaknesses to bypass supervisory controls in the system. Consequently, security control in the operating system has to be rigorous.

13.3.1 The Trinity of Troubles

We emphasized earlier that security is a difficult problem. Then, the question that arises is why. A partial answer is as follows. Although we all are familiar with the term computer security, a suitable description or a formal definition of computer security is problematic. Simply put, the goal of security is to assure the users that the system will behave appropriately. However, what is considered appropriate behaviour is system-dependent. The description of security is particularly difficult for any complex system and most computer systems are indeed very complex. The complexity arises mainly due to software components that distinguish computer systems uniquely from other technological innovations. So essentially, computer security is primarily a software problem.

Versatility of software to do almost anything has made computing ubiquitous and most software systems designed for practical use are inherently complex. Computer security is about making its software behave appropriately. Three factors make this task increasingly difficult nowadays; they are, known as the *trinity of troubles*—complexity, extensibility, and connectivity.

≫ Security and functionality are two often competing and conflicting goals. Higher functionality prefers greater flexibility in accessing the system and higher security demands restricted accesses. Achieving balance between the two is the challenge in designing protection for a system and must be thought thoroughly at the inception.

- **Complexity**: Software systems acquire complexity quite easily. The simplest example is the Windows operating system, which evolved from less than 5 million lines of code in 1990 to more than 40 million lines in less than 15 years. More lines of code imply potential for more bugs and, hence, a greatly magnified chance of undesirable behaviour of the software.
- **Extensibility**: Most modern computing systems are extendable and, hence, evolve over time. For example, many operating systems support dynamically loadable modules and drivers. Most software accept updates or extensions from remote hosts mainly through mobile codes. Mobile codes are powerful entities and form one of the foremost vehicles for launch of attacks on security in computer systems.
- **Connectivity**: The growing trend of Internet connectivity renders computers vulnerable to remote attacks. Further, attacks on or failures of one system easily spread to other systems.

≫ The need for more functionalities and the potential for extensibility and connectivity cause a system to evolve. Most complex systems in nature are evolutionary and future software systems are expected to be evolutionary systems.

13.4 Security Environments

We examine and analyse security from various angles to gain a deeper understanding of the subject.

13.4.1 Software Security vs. Application Security

As made obvious in the earlier sections, computer security is about first building secure software and then protecting that software from abuse. The first part is referred to as *software security* and the second as *application security*.

Software security is related to its design, construction, and testing. Secure software must behave to meet design specifications and should withstand attacks proactively. Weakly designed software is vulnerable to attacks on its security. The weakness may be system configuration problems, program bugs, or flaws in the program or system design. These bugs and flaws in the system constitute the *system vulnerabilities* that an attacker can exploit.

Application security is concerned with protecting the software during operation. It is chiefly finding and fixing known security problems. In a way, software security is to proactively assure security whereas application security is a reactive approach.

≫ Software security may be considered as an approach to design-, implement-, configure-, and use software such that the system functions correctly under malicious attacks of any form.

≫ In the software context, the term bug refers to a coding defect whereas a flaw refers to a more serious problem present at a deeper level. Typically, a bug is an implementation level problem and a flaw is a design level problem.

13.4.2 Internal Security vs. Boundary Security

From the perspective of the system, the problem of ensuring security may be divided into two parts: (1) ensuring security within the system and (2) ensuring security along the boundary of the system. The first part deals with the security issues related to the behaviour of the active entities (processes) within the system and the second with who (data, software agent, or user)

may enter the system from outside. We refer to the first part as *internal security problem* and to the second as the *boundary security problem*.

13.4.3 Malicious vs. Incidental

Security issues arise mainly due to two undesirable behaviours: (a) *malicious* and (b) *incidental*. Malicious behaviour attempts to read or destroy sensitive data or disrupt the system operation intentionally. It is usually initiated from outside the system to deliberately harm the system or the users of the system. Here the target under attack is the software and it is referred to as *target software*. The target could be a file, a server, or a system responsible for operating real-time systems. The software code designed to attack the target is known as *subversive code* or malware and various forms exist.

Incidental behaviours need not be intentional, and usually arise from within the system, but their consequences are as serious as those of malicious behaviour are. They may be due to hardware malfunction, or undetected errors from applications, operating system, compiler, etc., or damage from natural disasters or accidents.

13.4.4 Hardware vs. Software

Computer systems consist of hardware- and software resources. Security attacks on a computer system cause harm to one or more of these resources. Hardware resources include processors, the main memory, secondary storage devices, and communication lines and devices. The control of the processor wrested illegally and the services denied to other processes in the system. Illegal access to the main memory or secondary devices can lead to loss, destruction, or unwanted change of data. Illegal access to communication lines can have the most serious effect on the system. Not only can data be stolen or altered, but also any service or user can be easily impersonated and further access gained to the data in the system (both in the network or inside the system).

In software, data and service (process) are subject to security risk. Data include data files, programs, and various system components such as the page table, stacks, queues, or communication ports. Data can be read, modified, leaked, and/or destroyed. Unauthorized information leaks (also called *information disclosure*) relate to illegal dissemination of private information. This easily compromises privacy and secrecy and can result in substantial loss to an individual, group, or organization. Destruction- or alteration of information without dissemination has similar effect except that it need not compromise privacy and secrecy. Two important qualities of data are their *confidentiality* and *integrity*. Data confidentiality preserves secrecy of secret data, and unauthorized leaks might compromise it. Data integrity assures users authority on its modification. Unauthorized data alteration compromises the integrity of system data.

Security of a service or a process may be compromised in many ways. A service or process may be illegally used, invoked, influenced to behave in a certain way, proliferated, or destroyed.

13.4.5 Unauthorized Use vs. Denial of Service

From the users' point of view, security abuse may be classified broadly into unauthorized use of service and denial of service. Unauthorized use of service is broad and includes activities such as using another person's computer to check mails, using another person's account to gain access to the system, making copies of copyrighted software, etc. On the other hand, denial of service is intentional prevention of authorized users obtaining their services on time. This is achieved at many levels including completely preventing access to the system, slowing down the system by flooding it with unwanted processes or messages, bringing down the network, etc. Whatever be the reason, any service not provided within an acceptable period is called *denial of service*. A related concept is *system availability*, which in this context means that no security breach can make the system unusable or unavailable.

13.4.6 Inside Attack vs. Outside Attack

Security attacks could be either from within the system or from outside the system. The attacker within system may be an unauthorised user or a malicious process. When an unauthorized user gains access to the system either using an authorized account or through the loopholes of the operating system, processes with access rights can be established in the system to abuse its privileges to access the files and other resources. In addition, such processes are used to find more security weaknesses in the system. A *malicious process* is an illegitimate process in the system introduced or created from outside the system using weaknesses in the security of the system and performs the illegal tasks as directed by its creator. A malicious code or process that works for a remote user is called an *agent*. In essence, a system can be attacked from inside by directly creating a regular process with access rights through an authorized user account or indirectly from outside by a software agent.

As the Internet is becoming part of everybody's life, computer attacks from outside are becoming increasingly common. These attacks occur through network channels mainly using messages. The attack could be through *legitimate channels* or through *illegitimate channels*. In the former case, the user may hold an authorized account in the remote computer connected to the network and has legitimate connection to the system under attack. Here the user may simply misuse the network and system services to choke the system or steal information. In the latter case, the user does not have a legitimate connection to the system under attack but manages to establish one by exploiting the security weaknesses existing in the system.

13.4.7 Security Attackers

In the majority of the cases, processes or messages created for illegitimate purposes do harm to the system and hence breach security. Human users, often with criminal intentions, create these processes or messages. These

security attackers are termed *intruders* or *adversaries*. The level of their aggression or intentions may vary from simply reading an unauthorized file to making changes to crucial data (e.g., illegally transferring money from one bank account to another account). Some of these people are experienced programmers who are well versed with the weaknesses of the system and know the ways to break into it. Such highly proficient programmers are called *hackers*. Although, in public, hackers are viewed as unscrupulous, in the world of security they are considered highly respected programmers. Most programmers who write security software may be considered hackers. To distinguish them from good hackers, hackers with dishonest intentions are called as *crackers*. In this book cracker, intruder, and attacker mean the same.

» "I use the term *hacker* in its original, positive sense. I also believe that all good virus researchers are hackers in the traditional sense. I consider myself a hacker, too, but fundamentally different from malicious hackers who break into other people's computers."
—Peter Szor, The Art of Computer Virus Research and Defence

Humans by nature are curious. When accessibility is easy, many people are usually tempted to read others' files and electronic mails. Some mail systems can remember the login id and password so the user need not enter the information explicitly every time to login into the system. This facility allows anyone to access such systems without knowing the user account information. (Directories and files in most UNIX systems by default are created with public read permission.) These people are intruders, but they are the weakest group of crackers. Their activities include casual observation of others' e-mail on a computer screen or observing keystrokes of others. No sophistication in the computer technical knowledge is needed for this type of intruding.

The severity in cracking starts with highly skilled students, system programmers, or administrators whose intention need not be anything more than mere curiosity, entertainment, personal challenge, or adventurism. These type of crackers are usually *snoopy insiders* and devote substantial amount of time to such activities. These activities may include a sophisticated level of snooping using computer programs to monitor activity or collect information, track data movements and resource usage, etc.

» Although the term snoopy usually implies negative intentions, it is used in many positive contexts. There are many utility programs, snoop (a command line packet monitoring tool), snoop protocols (used to monitor TCP-aware link and provide reliability and congestion control, for example by resending lost message packets based on duplicate acknowledgements, suppressing redundant TCP acknowledgements, etc.), and snoopy servers (used to capture network traffic analysis) are designed to ensure effective system analysis and utilizations.

The intensity of cracking rises as motivation is elevated to making money. This group of crackers are usually skilled programmers and use varieties of tricks to get money. A typical example is modifying bank software to transfer money to a designated personal account. The modification may be to truncate the values of all- or selected accounts and collect the fractions and/or money from dead- or old accounts. These types of activities are hard to detect. Further, these individuals, sometimes, may go to the extent of blackmailing customers or the targeted financial organization to pay up, threatening to compromise confidential information or security. At the extreme, even groups or organizations may initiate these criminal activities. Commercial organizations may try to obtain information about its competitors and use it to their advantage. Military organizations may try to obtain information about enemy countries' sensitive defence information.

Crackers can either obtain information and use it for their gain or create destruction. These two objectives are usually achieved by executing software

programs in the target machine or on the network in which the target machine is connected. There are two fundamental ways of dealing with these issues, prevention and cure. Prevention generally involves disallowing such illegal programs to enter the system or the network and using secure communication. Secure communication means that no unauthorized person or program can access the information, and even if the data is accessed the original information cannot be extracted. Cure mostly entails destroying the illegal programs from the system and networks once their presence is noticed and repairing the damage.

With this overview on computer security, next we look at the possible security vulnerabilities of computer systems.

» A secure system must assure that no unauthorized person is allowed to access the information (confidentiality) or to modify the information (integrity), that the authorized persons must be able to get their services from the system (availability), and the authenticity of information must be assured (non-repudiation). In essence, confidentiality, availability, integrity, and non-repudiation are the main components of a secure system.

13.5 Security Vulnerabilities

Vulnerabilities are weak spots in the system that can be exploited by attackers. System vulnerabilities are classified as system bugs and system flaws based on the source and severity. Bugs may be introduced to the system through carelessness in coding and most of them may never be executed. System flaws are more serious and usually occur due to poor design. Error handling and recovery mechanisms that fail in an insecure fashion introduce common flaws. The flaws could be in a simple localized implementation or exposed when composite actions of many components interacting through their interfaces. The possibility of race condition is a classic example of such composite system flaw. Next, we look at three classic vulnerabilities that are heavily exploited by the attackers.

13.5.1 Buffer Overflow

A buffer is a storage structure used to hold data during execution. A buffer overflow can occur if the size of the information written in a buffer is greater than the size of the buffer itself. For example, string-handling routines in C library identify the end of string by NULL character trusting that the user or application will supply it. When the NULL character is not found, the size of the string becomes unknown and the problem explodes by overwriting the buffer, spilling over to its surrounding region. This can create various side effects in the system that the attacker may take advantage of. A buffer overflow can cause some critical system programs in memory overwritten and then, as a result, the system may crash. Hence, denial of service is a possibility due to buffer overflow. An attacker usually does much more than just making the buffer overflow.

We will see how the most common type of buffer overflow called *stack overflow* can be exploited to get the control of the system. Recall that an application process has program code, data, stack, and sometimes heap. The stack is used to store function call-by arguments, local variables, and values

of selected key registers. The heap holds the dynamic data structures. When a function is called, the return address for the function call—to resume execution after returning from the function, is placed on to the stack. This is the place exploited to acquire control of the system. If the attacker somehow knows the address of this location, say k, and makes the stack overflow by her input, then that input can be designed in a way that the stack overflow causes to put (a) her own code, referred as *payload*, into the target system's memory and (b) the address of payload at location k in the stack. Upon termination of the function, the address at k—which is the address of the payload program, is loaded into program counter and now the attacker gets control of the system through her payload program.

Buffer overflow can be easily avoided by checking the input data before actually placing them in the buffer. That is, if the inputs to the buffer are not legitimate they can be simply discarded.

≫ A typical example of buffer overflow exploitation is through a bug in the finger utility that runs as a daemon process under most UNIX systems. This utility can be invoked remotely from any system connected to the network. When invoked, the information supplied by the remote user is placed in the runtime stack of the daemon without checking its length. The finger utility may be invoked by supplying its parameters in such a way that it causes a stack overflow. In this case, instead of returning to its caller, the function returns to the address of the payload carefully crafted by the attacker as part of the input to the daemon process.

13.5.2 Trap Doors or Back Doors

A *trap door* is an undocumented feature usually created by the system designer or system programmer for developmental- or testing purposes. It could also be a system flaw vulnerable to attacks. The more common trap doors are purposefully implanted as privileges. Many such trap doors are typically inserted during development and testing to bypass the execution of certain segment of codes, verification of certain parameters, etc. Actually, such codes are inserted for convenience and fast testing, but it becomes a trap door if the code is not removed immediately after its purpose is over. Although most of the times trap doors are introduced for convenience during system development, sometimes trap doors are implanted to carry out illegal activities such as changing- or tampering valuable data in a database without authorization.

Inserting trap doors is a common- and desirable technique during design-, development-, implementation-, and configuration of large software systems such as operating systems. Since majority of the trap doors are inserted in an ad hoc fashion, identifying and removing them effectively is quite difficult. When a trap door is used to access the computer bypassing security checks, it is called a *back door*. Back doors may be installed by the programmer or by an attacker. The attacker may enter the system through a current flaw and install a back door for future entry, expecting that the current flaw of the system may be identified and fixed at any time.

≫ The simplest example of a trap door is inserting a code in the login program to allow an easily recognizable login name (say, 1234) to access the system without entering a password. If this code is not removed after testing, then anyone can enter the system by using the login name 1234.

13.5.3 Cache Poisoning

Many systems cache their clients' data locally and use them to serve future requests without verifying with the data sources each time. If an attacker poisons the cache with some incorrect data, the system cannot know that it is

serving an illegitimate client. Once an attacker becomes a trusted client, then she can exploit other system flaws to perform new attacks on the system.

13.5.4 Other Vulnerabilities

Security vulnerabilities are likely to exist always in large complex software systems and operating system is not an exception. The most common classes of vulnerabilities in operating systems are as follows:

- **Insecure Bootstrapping**: Any operating system acquires full control of the computer only after it is completely initialized. Therefore, the system is vulnerable before it attains full control of the computer. In addition, most operating systems can be booted with different level of access privileges.
- **Configuration Mistakes**: The default settings of most operating systems are not highly secured. The system not adequately configured for security is vulnerable to attacks.
- **Weak Authentication**: Only legal users are to be allowed to access the system. The most common authentication mechanism is the password and many password systems have the potential weakness to be broken or the password guessed easily. Weak authentications may allow illegal users enter the system with legal user privileges.
- **Weak Cryptographic Algorithms**: Cryptographic algorithms are used for both password encryption and secure communication (discussed later). Weaknesses in these algorithms are a major vulnerability of the system.
- **Improper Input Validation**: Buffer overflow and cache poisoning are examples of vulnerabilities due to improper input validation. Many popular- and widespread attacks are by exploiting these types of programming weaknesses.

≫ Domain Name Server (DNS) cache poisoning is a typical example. DNS has the flaw to accept data without verification. Exploiting this weakness, an attacker can send fake data to a DNS server. In this case, the DNS server can store the received unauthenticated data into its cache and serve the same request from the attacker in the future.

≫ System vulnerabilities may be reduced to some extent by methodical and careful system design, implementation, and testing, but can never be avoided altogether. As long as a system has its inherent vulnerabilities, it cannot be completely protected. Therefore, application security is a complementary approach to system security rather than an alternate approach.

We know that the operating system is the heart of any computer system. Avoiding these vulnerabilities of the operating systems will boost the security of the computing systems and the applications that they support. However, in practice, the task of avoiding vulnerabilities in any software system is much more difficult. Any complex system needs a preventive approach during its development phase, rather than the identify-and-cure approach during its operation. We have seen that system security is about designing, building, and testing the system for security and application security is about finding and fixing known security problems after the system is deployed.

A detailed introduction to system security and application security is beyond the scope of this book. We only look at some key aspects of security in computing systems.

13.6 Access-control-based Security Systems

Access control-based security systems are based on a simple and powerful model of computer system. A computing system may be studied for various aspects such as efficiency, security, fault tolerance, etc. Models often help to understand- or study the indented aspect of the system by their convenient representations. Such a representation of the system under study will expose only the essential feature of the system and hide unnecessary details. Here the computer system must be modelled for the purpose of security. One such security model is the *access control based model* that we study in this book.

In the access control-based model the computing system is abstracted as a collection of *subjects* (active entities) and *objects* (passive entities). The principle objective of the system security is to have a set of security policies, which describes how the objects are accessed by the subjects in the system. In this model, *access control policies* are the security policies. A security policy is a statement on what *privileges* and *limitations* a subject has on the object. The policy may be simple and generic to be independent of other state information, or it may dependent on how- and when the access is performed. For example, a policy could be so simple that a subject s has read and write access to an object x or so complex that the subject s can read the object x only when s has completed its write on another object y. This approach implements security by suitably protecting objects from the subjects. Here we use the term protection in a very generic sense, restricted it to controlling read, write, and execute operations. Such higher-level abstraction will help to visualize- and model various security options.

> Modeling is a convenient way of abstracting a system for a particular purpose. Here the objective is to model the computer system for the purpose of security. A model is simply a useful representation. For example, a computing system may be modelled as a collection of active- and passive entities. Processes are the active entities and resources are the passive entities, and active entities use or consume passive entities.

> The key elements of access control models are *subjects*, *objects*, and *access rights*. The entities in the system to be protected are called objects, the entities from which the objects are to be protected are called subjects, and the kinds of accesses that the subjects can execute on the objects are called access rights. A security policy establishes a relationship between subjects and objects, individually and/or collectively.

13.6.1 Object, Subject, and Access Control

Objects and subjects are the key components in access control-based security systems. Our objective here is to study how to protect system objects from its subjects. Here, the term object is used in a very generic sense. Objects could be hardware resources such as the processor, memory, devices, or software resources such as processes, page tables, files, semaphores, locks, etc. Sometimes we use the term object to mean the data it encapsulates and a predetermined set of operations that can be applied on the object. Different objects support different operations. Operations are the only means to manipulate data contained in objects. An *access* or *reference* to an object means applying exactly one operation on the object. For example, open, read, write, append, close, etc., are valid operations that are applied on files. In this respect, protection is a measure of confidence that the integrity of the operating system and its objects will be preserved.

> Objects may be viewed as abstract data types. This view gives uniformity by defining well-defined operations, states, and access rights for each object and hiding the physical and implementation details from the subjects and other objects.

Where protection is concerned we represent a resource (or any piece of information) by a passive entity called *object*. A system contains many objects manipulated by active entities called *subjects*. Subjects apply operations on objects. Subjects can also be objects in our system, in which case a

» Examples of objects are process, file, CPU, memory, etc., and examples of subjects are process, thread, computer user, etc. A process can be both a subject and an object. For instance, in UNIX, a process can apply a **kill** operation on another process.

subject can manipulate other subjects. Not all subjects are allowed to access all objects because some subjects may not be completely trusted. Unrestricted accesses to objects by subjects are undesirable too and may become a potential risk to system security. For example, an ordinary user must not be able to modify system utilities. Most subjects need to access only a small fraction of objects to carry out their tasks. Thus, we need a guard to control the use of each object by each subject. That is, there must be well-defined rules to govern the manipulation of objects by subjects.

Every subject is associated with a limited view of the computer system. The view describes what objects the subject can access. In addition, a subject may not be permitted to execute every valid operation on every object in the view. The view also identifies, for each object, a subset of operations that the subject can apply on the object. That information defines the security-related 'authority' the subject has in the system. We need two things to ensure protection in the system: (1) the access control or authority information must be available to those guards that supervise objects, and (2) each subject must have a reliable way of identifying herself to the guards. Access control defines these rules for grant or denial of certain permissions to access the resource. Access control is the heart of this security system.

Different security systems employ different access control mechanisms for protection purposes. Loosely, an *access control* mechanism is a way to control information from objects accessible to different subjects. This is done in three steps: (1) For each subject, the system establishes her authority to access various objects. The system must have access to information pertaining to each subject's access rights (rights to apply operations on objects). (2) The system provides an environment in which subjects can access the authorized objects, but not unauthorized ones. (3) Every access made to an object by a subject is scrutinized by the system before the subject can actually access the object. The system must be capable of using the subject's access right information to control its accesses to objects. The system needs to make sure that subjects do not cross their authorized boundaries.

» Although checking each access to objects slows down program execution speed, it definitely improves system reliability. Such checking helps in early detection of unknown hidden errors in applications, especially malicious ones.

In addition to protecting objects from subjects, a protection system must be able to protect itself; that is, protects its own programs and data. For example, a user program must not be able to flip the processor-operating mode while in the user space except making system calls. Another example is of the operating system protecting itself from user processes.

» Open environments are susceptible to design errors because a forgotten restriction will enable an unauthorized action. In contrast, in closed environments a forgotten permission will prohibit an authorized action, which is better than permitting unauthorized actions.

Generally, a protection scheme provides one of the two following alternative environments: In a *closed environment* an action by a subject is not permitted unless the subject is explicitly authorized. By contrast, in an *open environment* an action by a subject is always permitted unless the subject is explicitly not so authorized. Although the environments are functionally equivalent, most operating systems provide a closed protection environment because it is more error-resistant. The fundamental principle of a closed environment is to give each subject no more authorization than needed to perform the tasks. For the purpose of protection, the operating system employs separate control objects (that are different from the objects

proper under protection). For example, in UNIX systems, to protect a file, the file management system maintains additional read-, write-, execution permission bits in the file metadata. The operating system needs to protect these control objects too.

13.6.2 Access-control Models

Once the higher-level access control abstractions are established, the next question is how the access control is defined or who imposes it. To answer this question, we discuss three popular models of access controls, namely discretionary access control (DAC), mandatory access control (MAC), and role-based access control (RBAC) introduced in literature.

- **Discretionary Access Control**: This model relies on the object owner's discretion to control accesses on that object. It is the most popular approach implemented in most operating systems including UNIX. This is user-driven access control and is local to individual users.
- **Mandatory Access Control**: This model advocates for system-wide policies on access control, usually set by system administrators and enforced for all users. This model complements DAC in protecting system-wide sensitive information. The model is useful particularly to control *information flow* between subjects. Controlling information flow between subjects is very difficult.
- **Role-based Access Control**: This model is more complicated than the other two. In this model, users are assigned to different roles time to time by a security administrator based on their positions, responsibility, and competency. Access control is then defined based on the roles of the users. A role simply defines the least required access privilege to perform the job in that role. For example, an administrator needs more privileges than an ordinary user.

The access control system is a dynamic system where the state changes from time to time. The *state* of an access control system is a set of subjects, objects, and access control configurations, which determine the authorization relations for subjects on objects. Among the three models DAC, MAC, and RBAC, DAC is most popular and is used in almost all operating systems.

13.6.3 Discretionary Access-control Model

In discretionary access control model, every object is owned by some subject, and owners of the objects enforce the control of accesses to objects. The control is defined between individual subject and individual object. Consider that a system has a set O of objects and a set S of subjects. Then, the most natural way of representing (abstracting) the access control is by a matrix,

called **access control matrix** (ACM) or **protection matrix**. The matrix contains the information about the subjects and their respective rights on respective objects (see Fig. 13.1). Each subject is associated with a row in the matrix, and each object a column. For each subject and object pair, the corresponding entry in the matrix cell defines a set of access attributes or privileges that the subject has for the object. Typical attributes are 'read', 'write', 'execute', 'open', 'close', 'connect', etc. In addition, flags or signs are used to represent ownership of an object. Each object is owned by a subject and she may grant and revoke access privileges on the object to other subjects. We say a subject has *x*-access right on an object if the corresponding matrix cell contains the *x* attribute. For example, in the protection matrix of Fig. 13.1, subject S1 can execute object O1, but cannot read or write the object.

» Empty cells in the access-control matrix are interpreted differently in different security environments. In the closed environment, the subject does not have any rights on the object, but in the open environment, the subject has all rights on the object.

For a given protection matrix, all the cell in a row collectively define a *protection context*. Similar to the concept of process execution context, a protection context precisely identifies the rights the subject has on objects. The protection context identifies only those authorized or permitted operations the subject can apply on which objects. Every other action by the subject is an unauthorized operation. (Note that we are considering the closed protection environment.) When a subject is assigned a protection context, she inherits the power associated with the protection context. Different protection contexts may have different powers.

The content of the protection matrix represents the current *protection state* of the system. Certain operations by subjects can alter the protection state. That is, a protection matrix is not a fixed object, and is changed by adding or removing new subjects and objects. For example, in the protection matrix in Fig. 13.1, subject S2 can delete object O1. If she does so, column O1 will be removed from the matrix. In addition, subjects may alter contents of particular matrix entries: some access privileges of subjects can be revoked, altered, or added. For example, in UNIX systems, the owner of a file can grant or revoke another user (of the system) a permission to read the file.

Objects →

Subjects	O1	O2
S1	Execute	Read
S2	Delete	Read Write

Figure 13.1: *A typical protection matrix.*

The ACM itself is an object created in the system for the purposes of protection and security and must be managed securely. The concept of access control can be generalized to apply beyond simple objects and subjects. Not only objects such as file and data must be protected from subjects such as processes and users, processes must be protected from one another. Therefore, access matrix could also be created for subjects vs. subjects, particularly processes vs. processes. For example, a process can interact with another process by *sending* and *receiving* messages, and *blocking* and *unblocking* one another for synchronization purposes. In this case, we have four operations, send, receive, block, and unblock, and they can be specified in the access control matrix to provide necessary security on process interactions.

Protection Domain

Providing access control at the individual subject level may not be always desirable for many reasons. First, if the system has a large number of subjects and objects, the size of the ACM becomes very large and sparse and, therefore, not efficient for implementation and management purposes. Secondly, this scheme does not reflect the reality of many organizations. It is normal in practical life that groups, classes, and categories of users have the same rights on some common objects. This is true to some extent in the computing world too. Those subjects who share common attributes and states can be grouped and assigned the same access rights. To deal with such realities, the subjects with the same associated rights may be grouped into a logical domain and known as *protection domain*.

≫ The domain concept adds one more level of abstraction, and helps classify a large number of subjects into a fewer numbers of domains.

Each row in a protection matrix can be considered a protection domain. Actually, domain is a better term, and it decouples a protection matrix from subjects. Domains establish the foundation of the protection system in modern operating systems. These systems allow many subjects to operate in the same protection domain. Each domain defines constraints within which all its subjects may access certain objects. (For our purpose, we assume that domains are the computer user accounts or groups, and subjects are processes the users create.) One added advantage in this model is that subjects can switch form one domain to another, i.e., move from one protection context to another.

Each domain defines one protection context within which all its subjects operate. The context identifies which subset of objects is available to the domain and which subset of operations can be applied on those objects by the domain subjects. A subject operating within a particular protection domain must restrict herself to the objects available in the domain. For example, in UNIX systems, each user is associated with a protection domain identified by her user and group identification numbers. Every process owned by the user has an identifier called user identifier (*uid*). The operating system uses the uid to determine what access rights the process has on various resources. Another example is of the processor hardware implementing two protection domains depending on the value of the processor-operating mode. In the user-operating

mode, certain instructions and resources are not available to the processes. If a process tries to use them, the hardware generates exceptions and the operating system traps the process. However, in the kernel-operating mode, a process can execute any instruction and can access any resource. The two modes help protecting the operating system from application processes.

In some systems such as UNIX, the superuser is a special user who has all access rights to all objects. Other users, called ordinary users, have lesser privileges. In these systems, all processes belonging to a particular user operate in the same protection domain of the user. All processes have the same access rights to all objects that the user has. Any process that modifies its domain composition immediately affects all other processes of the user. Some systems define multiple domains for a subject. Different processes of the subject may operate in different domains. Modifications in one domain do not affect processes in other domains.

≫ A set of objects associated with some access rights can be grouped to form a protection domain. Rings of protection domains, introduced in MULTICS, is a classic example of protection domains. In MULTICS, the innermost ring has the highest protection; the next ring has the next highest protection and so on; the protection level decreases as the direction moves outward. UNIX may be viewed as having two rings: kernel mode and user mode.

The concept of protection domain is very useful in structuring the system for security purposes. Processes can move from one domain to another with different access rights in different domains. Protection domains can overlap, giving different access rights to objects in different domains. Processes can *enter* and *exit* domains. So, a protection matrix may be constructed to control switching between domains by entry and exit operations or by a simple *switch* operation.

Domain Switch

As mentioned previously, each subject operates in a single protection domain at any time. The domain defines the limits of the subject's rights to objects. In static systems, a subject cannot change its domain. In dynamic systems, a subject can change its domain. Some operating systems support a limited form of domain switching. Domain switching is a switch-operation on an object that itself is a domain. For two domains d and d', if the corresponding protection matrix cell for the row d and column d' contains a switch-operation, then only a subject operating in domain d can switch to domain d'. In Fig. 13.2, subjects from domain *D1* can switch to domain *D2*. When a subject does so, it has different access rights on O1 and O2. For example, it can no more execute O1 as long as it is in D2.

≫ The switch is the only operation between two domains.

In UNIX systems, each user is associated with a protection domain. A process normally operates in the protection domain of the process owner. The process owner's user identifier (uid) determines the current domain. One way of switching a domain by a process corresponds to changing its user identification temporarily. The uids are unforgeable authentication identifiers, and UNIX systems do not permit changing process uid attribute. They instead use another identifier, namely effective user identifier (*euid*) that is actually used to determine the process's current protection domain. Normally, for a process, euid is the same as the uid. A process cannot arbitrarily change the euid value. Only certain system operations can change the value; without such restrictions any process can assume the role of a superuser process, and do anything it likes. Each executable file has a boolean domain attribute in the

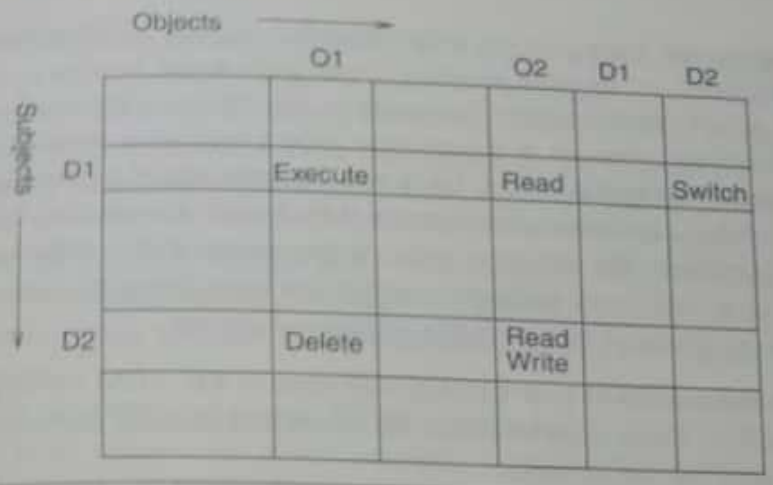

Figure 13.2: *Protection matrix with domain switch.*

file metadata, called *setuid* bit. If the setuid bit of an executable file is set, any process executing the file will operate in the domain of the file owner, and not in the domain of the process owner. That is, the process euid becomes the uid of the file owner. These systems allow euid of a process to be changed to that of the program file owner when the process executes the program, and thereby, changing its protection domain. The execution right of the process owner on the program file decides the permission for the domain switch. When the program execution is over, the process reverts to the old domain (another domain switch).

Another example of domain switch is the complementation of the processor-operating mode. Modern processors implement specialized instructions for this purpose. For example, a system call changes the processor from the user-operating mode to the kernel-operating mode. When the system call returns, the processor reverts to the user-operating mode.

In summary, access matrices are constructed between:

- subjects and objects (e.g., processes vs. files, users vs. files),
- subjects and subjects (e.g., processes vs. processes),
- objects and domains (e.g., files vs. domains), and
- domains and domains (e.g., user mode vs. kernel mode, ordinary user vs. system admin).

Access control matrix is an abstract representation. Next, we look at how the access control matrix is implemented in real systems.

13.6.4 Implementing the Access-control Matrix

Access control matrices are typically large and sparse. In addition, objects, subjects, and domains may be created and destroyed dynamically. Therefore, for efficiency and convenience, access control matrices must be implemented in a distributed way using dynamic data structures, and not as a static

two-dimensional array. To get a better understanding of some implementation techniques, let us have another closer look at the matrix.

The access control matrix has columns for objects and rows for domains. Each column corresponds to a particular object indicating what *access rights* each domain has on that object. Each row corresponds to a particular domain indicating what capability or privileges that domain has on the objects in the system. Therefore, the columns indicate the access rights from the objects' point of view and rows indicate capabilities from domains' point of view. This analogy gives us two possible implementations:

- **Access-control List (ACL):** Columns of the ACM are implemented as lists. Each element in the list contains a domain and a set of access rights for the domain.
- **Capability List (CL):** Rows of the ACM are implemented as lists of capabilities. A capability is thought of as a pair $\langle o, c \rangle$, where o is an object name and c is a set of privileges or access rights on the object.

Although these two implementations are equivalent, in general, they do not offer the same level of convenience and efficiency. For example, it is easy to determine what domains can access a given object using ACL and, therefore, easy to revoke all those rights to protect that object. On the other hand, using a CL, it is easy to know the totality of access rights of the given domain and, therefore, easy to revoke all the access rights of that domain. Most systems use both ACL and CL.

≫ ACL is used by most mainstream operating systems. Although many pure capability-based operating systems were developed, none is available commercially. UNIX implements both for different purposes. Owner-, group-, and other users, and read-, write-, execute permissions associated with each file are examples of access lists. File descriptors are examples of applications of capabilities in UNIX. The runtime file descriptor table is a CL in UNIX. When a process opens a file, the capability of the process on the file is updated in terms of its file descriptor.

Access-control List

For each object, we associate a list of domains from which subjects may access the object and for each domain, we specify a subset of operations that the subjects can apply on the object. That is, each object has its own ACL that is solely used to control accesses to the object by subjects from different domains. Thus, an ACL for an object is a list of ⟨domain, right⟩ pairs. It represents one column of the protection matrix. When an operation o from a subject belonging to a domain d is applied on the object, the object's interface guard searches down the list for the existence of the pair $\langle d, o \rangle$ in the list. If a match is found, the operation is applied on the object. Otherwise, the operation request is denied. To access an object, each subject needs to carry a unique unforgeable domain identifier. For example, in UNIX systems, each process carries uid that is a system data and the process cannot modify the data.

≫ A capability acts as a token authorizing the use of the object to which it refers. A capability is used as a special identifier or address for an object such that possession of the capability confers a single access right on the object.

Capability List

For each domain, we associate a list of objects that subjects from the domain may access and for each object, we specify a subset of operations that the subjects can apply on the object. An ⟨object, operation⟩ pair is called a *capability*. A CL is merely a list of capabilities. A protection domain is

represented by listing all the capabilities in a capability list. It represents one row of the protection matrix, and includes all the capabilities of the domain. Each capability identifies an object accessible to the domain and one operation that subjects from the domain may apply on the object.

Capability lists themselves are valuable resources that must be protected from abuse by subjects. We can store capability lists as separate objects independent of the main objects under protection. A capability can only be created by the system, and not by users. All capabilities are maintained in the system space so that subjects of one domain cannot forge capabilities from other domains.

There is a centralized guard. When a subject from a domain applies an operation on an object, the object manager consults the centralized guard. The guard, in turn, consults the subject's current capabilities, and grants the access only if the subject's domain has the corresponding capability in the CL.

> Though a capability list is associated with a domain, the list is not directly accessible to subjects from the domain. Capability lists are protected objects and are maintained by the operating system. Modification of a CL by a subject is not allowed.

Merits and Demerits, and Composite Systems

Both CL-based and ACL-based systems allow controlled sharing of objects. The two schemes are functionally equivalent. The authorization mechanisms however are different in the two schemes. In the ACL, a domain is authorized to use an object by having its name placed on the object's user list. In capability-based systems, a domain is authorized by giving it a capability or ticket for the object. In ACL-based schemes, every object has its own guard that examines each access request coming to it and decides whether to entertain the request. The request is first authenticated to identify the domain from where the subject has issued the request, and then the request is authorized only if the domain has the right to perform the requested operation on the object; each request must carry some additional piece of information called *credential* that would help identify the subject's domain. Thus, the request processing is slowed down by the authentication- and access control search overheads. Capability list-based systems seem to run inefficiently too: Individual object manager needs to check with a centralized guard whether the subject has the required capability.

> ACLs and CLs are equivalent in the sense that they are different ways of materializing the same protection matrix.

Searching an ACL on every access request can be prohibitively detrimental to system performance. In addition, in ACL-based systems, for a given domain, it is time-consuming to determine what rights the domain has on various objects because the rights information is distributed among objects. In contrast, the access rights information for each domain is localized in capability-based scheme. However, alternation or revocation of permission to materialize by an object's owner will have high overhead: the owner may have no way to find where capabilities for the object are stored. Access revocation is a serious problem in capability-based systems. Such is not a high overhead operation in ACL-based systems. Capability duplication and distribution by subjects, without permission from object owners, may create protection violations in capability-based systems.

≫ The temporary capability is for the subject and not for the domain. If the domain loses the access privilege after the creation of the capability, the subject continues to make the same reference on the object without any hindrances.

Capability lists enable efficient use of capabilities, and ACLs enable efficient rights administration. Most practical systems employ both of them for different purposes. Such systems normally maintain ACLs. When a subject accesses an object for the first time, the corresponding access control list is consulted to determine whether the access is permitted. If not permitted, access is denied right away. If the access is permitted, the system creates a "temporary" capability for the subject. The capability token is retained by the system, and the subject gets a reference of the capability. In subsequent accesses to the object, the subject presents the capability reference to the system, and the internal capability is used for quick verification of access rights without consulting the ACL. For example, in UNIX systems, for each file, the file management system maintains a set of protection bits for three categories of users, see section 11.8 on page 316. When a process opens a file, the file management system creates a runtime capability (per process open file object) by consulting the access permission bits of the file. The system maintains the capability, and returns a file descriptor (which is a reference to the temporary capability) to the process. In subsequent accesses to the file, the process presents the file descriptor to the file management system. The corresponding capability is obtained using the file descriptor. This capability is verified by the file management system without cross-checking the original file permission bits. Different processes can have different open file objects for the same file. For every subject operating in a domain accessing the same object, there will be a distinct capability for the subject. In other words, a somewhat static access control data is kept in ACLs, and capabilities are used as temporary runtime objects for fast authorization checks.

≫ Most computer systems have different types of resources. Accesses to resources may not be controlled at a central place. For example, access to memory frames are controlled by memory management hardware, access to files by one or more file management systems, and so forth. Capabilities of a subject are grouped into different classes, and these classes are placed in different CLs according to their use.

Whatever be the mechanism used to implement a protection scheme, we need to protect the protection information itself. The operating system mediates references to protection information; it does the mediation when subjects access objects. The mechanism of the mediation is called a *reference monitor*. The monitor rejects any accesses (including improper attempts to alter protection information) that are not allowed by the current protection state and protection rules.

13.7 Authentication

A major challenge for an operating system is to distinguish legal users from the illegal. It is called the *authentication problem*. The effectiveness of security in a computer system depends on the authentication scheme the operating system employs. Authentication, in general, is a way of establishing the identity of one party to another. In the operating system domain, it is a mechanism that allows processes and resources (used by the processes) to verify one another's identity. At the outset, there is a need for the operating system to determine whether a user is legally authorized to use the system. Only when a user is identified correctly, the operating system can apply local protection mechanisms on all operations the user performs. A user who wants

To use a computer system needs to identify herself to the system first. The question is how does the system authenticate her identity? As stated earlier, a piece of information, usually called the credential, is used to authenticate the identity of a user. Passwords, certificates, smart cards, biometric signatures (such as finger print, voice print), etc., are often used as credentials. When the operating system requests for credentials, the user presents her credential to the system, and the system authenticates the validity of the credential. The security check is as simple as that. The check can be performed on each request from the user, or once at the beginning of her new session with the computer system.

13.7.1 Password-based Authentication

Most commonly used authentication scheme in computers today is that a user identifies herself to the system by means of a login name and a password. The *login name* is a text that is her identification to the system. The *password* is a secret text that is supposed to be known only to her. All legal users have distinct login names. Note that login names may be publicly available, that is, those names may not be secret.

When a user attempts to open a new session with the computer system, she informs the system her login name and the secret password. The system examines the presence of the ⟨login name, password⟩ pair in a password file maintained by the system. If a match is found, she is considered a legitimate user of the system, and is allowed to start a login session. As noted earlier, the system normally maintains a unique numeric user identifier, *uid*, for each authorized user. The login process sets the uid and executes a command interpreter such as shell.

A login request may fail because either the login name or the password is not recognized by the system, and it is called an authentication failure. Most systems implement a single sign-on policy in which once a user who is authenticated can use system resources without further authentication. Once logged in, the login name is no longer used for authentication; the uid is sufficient. The uid alone identifies the authorized protection domain for the user. All the processes the user creates carry the uid value, and the value is not a forge-able entity.

Some systems associate passwords to sensitive files. Whenever users open such files they must supply the corresponding passwords. Different files may have different passwords that are different from login passwords. This password scheme is enforced by the file management systems, and not by the login authentication system.

Password-based authentication is simple, but comes with its own defects. The mentioned defect lies in the choice of passwords: which passwords are more secret. In remote terminal-based systems, passwords may get exposed to the outside world during communications between a terminal and the computer. It is a one-way authentication scheme; it authenticates users to computers but not vice versa. An intruder may reroute communications from

≫ In distributed systems, authentication forms a foundation that enables diverse local security policies to integrate into the global framework. There, every request from users (their application processes) to remote resources may need authentication. Non-distributed systems mostly provide session-based security, and the user is authenticated before starting each session. The protection system will take care of the session until the user closes it.

≫ The login name and password pair controls the login to the computer system, and the uid value controls accesses from the user to resources.

≫ There are many variations of this password based basic authentication scheme. For examples, passwords may be good for one use, and a user carries a list of passwords, striking out each one off the list as soon as she uses it. Passwords may have an expiration date, at the end of which users are forced to change their passwords.

user terminals to her computer. We discuss password maintenance and security in communications in the next two subsections.

13.7.2 Password Maintenance

≫ A research study on UNIX password files revealed that one-fourth of the passwords could be guessed with ease.

Login names are passports and passwords are visas to enter a computer system. Passwords are highly secret information. It is the duty of the operating system to keep passwords top secret. Password information is not to be disclosed to unauthorized users. An ordinary user should not be able to obtain passwords of other users.

Many a time users have simple passwords that are easy to remember. Passwords such as name of birthplace, residence town, school, spouse, pet, etc., are most common. An intruder may socialize with a user and easily obtain such information from her. Users should use "difficult to guess" passwords containing both alphanumeric and non-alphanumeric characters. Another way to break a password is the brute force approach by trying all possible combinations of valid password characters. It would not take much time to break small passwords by the brute force approach. To make the brute force difficult, users should use longer passwords. In addition, users should change passwords frequently.

≫ For the operating system, it may not be possible to keep passwords top secret. It is generally not possible to ensure total security. Malicious users (intruders) may be able to guess the password of a user, and break into the system pretending to be a legitimate user. Nevertheless, protection and security subsystems must make security breach a difficult and costly task.

Passwords are extremely sensitive information, stored normally in a password file. Even if an intruder obtains a password file, she must not be able to derive individual passwords. In UNIX systems, passwords are stored in /etc/passwd files in encrypted forms. An encryption is a function that transforms an input text into a cooked (illegible) output text. It is simple to compute, but very hard to compute its inverse. That is, given a password p, it is easy to compute its encryption $e(p)$ for a given encryption function e, but given $e(p)$ it should be impossible or extremely hard to get back p. When a user types her password, the system transforms it by applying the same encryption function, and compares this transformed password with that stored in the password file.

13.8 Secure Communication

In networked systems, computers exchanges messages among themselves. These messages are sent through public transport networks. Messages may be removed, duplicated, or altered while in transit. Consequently, providing security to messages is a daunting task. Expert criminals/eavesdroppers might intercept messages, reroute them to different destination, alter message contents, and so forth. For example, in bank transactions, messages may contain account information; expert criminals can steal money out of the bank without any trace by altering messages. We have to safeguard messages and their contents from disclosures and alterations.

Protecting networked systems includes protecting the nodes and protecting communication links. Protecting the communication is achieved by encryption and decryption mechanisms. Data encryption is widely used to safeguard message contents as well as data stored in nodes. The encryption system scrambles data in messages before they are sent out of computers. Encryption is essentially

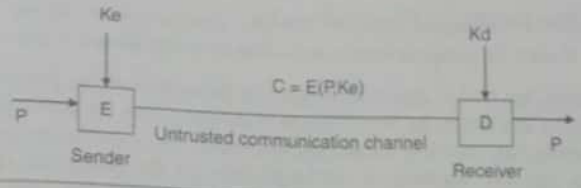

Figure 13.3: *The principle of encryption.*

transforming the message (called plain text) into secret unintelligible form called *cipher text* and decryption is the reverse process of transforming the cipher text into the original plain text. Encryption maintains secrecy by hiding information. The information or values used to alter the information from original to cipher and from the cipher back to the original, respectively, are called encryption- and decryption *keys*. Similarly, the transformation functions used for encryption- and decryption, respectively, are called encryption- and decryption functions. The theory behind this encryption and decryption is referred to as *cryptography*.

Consider that P is the original message, E is the encryption function, Ke is the encryption key, C is the cipher text (see Fig. 13.3). Then, C = E(P,Ke). If D is the decryption function and Kd is the decryption key, then, P = D(C,Kd) = D(E(P,Ke),Kd).

If the sender only knows E and Ke and the receiver only knows D and Kd, then the communication using encryption and decryption can assure secrecy, integrity, and authenticity of the message. In some cases, E = D and Ke = Kd called *symmetric key system*. In secure communication, the critical part is the key distribution. Keys must be distributed before any communication occurs in an un-trusted channel. One way of distribution is by sending the keys by courier service. This will be complicated and costly, if the communication involves many processes. The popular solution to this problem is called *trusted server* approach. In this approach, only the key for the communication with the server is obtained through outside system such as the courier service. Once the secure communication is established with the trusted server, then the keys for the communication between any pair of processes are obtained from the server by secure communication with it.

A popular cryptosystem is the *public-key cryptosystem* which uses *asymmetric keys*. There are two keys: public and private keys. The idea underlying the public-key based cryptosystem is that if one key cannot be derived from the other using any computationally feasible manner then one of the keys can be made public. The well known public key cryptosystem is RSA and it works as follows. The encryption- and decryption functions have the following forms:

$$C = E(P) = P^e \bmod n$$

$$P = D(C) = C^d \bmod n$$

where the pair (e,n) defines public key and the pair (d,n) defines private key. These integers must satisfy the following conditions.

1. $P = D(E(P)) = (P^e \bmod n)^d \bmod n$.
2. d must be computationally infeasible to derive from e.

Again, the trusted server approach can be used to obtain private keys. The private key d must be very difficult to compute from the public key e. It is extremely hard and time-consuming to compute the prime factors for large integers and that hardness is exploited here to the advantage of RSA encryption systems. Anyone may intercept the message, but only an authorized recipient can decipher the scrambled cipher text to obtain the original plain text using a secret key in a reasonable amount of time.

We have discussed so far, in a sense, system security approaches. Next, we look at some aspects of application security.

13.9 Application Security

As we have discussed earlier, application security concerns protecting the system after it is built. That is, finding and fixing security problems on an ongoing basis. In this mission, the primary responsibility is to guard against malicious codes. First, we look at some of types of malicious codes and attacks that are commonly employed.

13.9.1 Subversive Codes

Subversive codes (also called malwares) are any codes designed specifically for attacks on security. Although there are many forms of codes, at the highest level, a subversive code essentially performs any combination of the following activities:

- *Data collection*: It consists of techniques such as keystroke monitoring and network packet sniffing and inside activities of reading data from files, memory, cache, etc. The collected critical data are then used for illegal- or undesirable purposes.
- *Covert communication*: Covert communication is unconventional communication to maintain secrecy. It may be legitimate (used predominantly by secret agencies) or illegitimate. In this context, we are concerned with covert communication used for illegitimate purposes. It is a way of obtaining or conveying information illegitimately by seemingly legitimate use of computers and communication resources. Covert communication is primarily used to extract sensitive data and use them to the attacker's advantage. A *covert channel* is a communication path set up for covert communication.
- *Command and control*: These activities involve altering the behaviour of the system, denial of system services, destroying resources, allowing remote control of the system, etc.
- *Stealth*: This activity mostly tries to hide system elements (data, processes, users, etc.). Hiding data or processes may be used

positively for security purposes or used negatively to disrupt or attack the system.

In any case, the objective of a subversive code is to disrupt or steal sensitive information from the system targeted. Typically, a subversive code enters the system, gains control, and achieves its goal. Such a code may be considered as having two logical components: (a) *payload*—the main program (as already explained in Section13.5.1 on Buffer Overflow); and (b) *bootstrap* or *injection vector*—to establish the payload and set up to acquire the control of the system. Next, we look at some subversive codes frequently used:

≫ A subversive code can be a simple or self-replicative program designed for malicious purposes. Trojan horses and logic bombs do not replicate, but viruses and worms do.

- **Viruses**: Malicious programs are designed for many different purposes. The program acquires the name because it has many characteristics of biological viruses. It is capable of infecting the systems that come in contact with it by *replicating* itself, needs a host program to live on, and can cause *damage* to the host system. Self-replication is the most important property of a virus (originally introduced by von Neumann with the idea of self-building automata) and has the potential to spread to other systems. Damage can occur and spread in several ways: instantaneously, synchronized to a specific time or event, or at random. Viruses usually spread through executable programs via removable media such as floppy disks, file transfers, e-mails, internet downloads, etc.

- **Trojan Horse**: This type of malicious program does not replicate itself but is imported into a system by a trusted program. It hides its malicious actions by masquerading as something useful or desirable. The key part is its ability to hide behind an unsuspected legitimate program. Once the malicious program is invoked in the context of a legitimate process, it gains all the privileges of the host process. If the process is a system process (i.e., invoked through system libraries or utilities), then the malicious code can acquire the system privileges and the situation could become quite serious. It is an indirect attack from within a system. The host program could already be resident in the system, or may come via e-mails or may be downloaded from the Internet as Java applets. Login prompt emulation for harmful purposes is a typical Trojan horse program.

≫ The name Trojan horse is derived from Greek mythology. Greek soldiers hid themselves inside a wooden horse the enemy were attracted to bring into their city. After covertly infiltrating the city through the Trojan horse, the invading Greek soldiers attacked the city.

- **Logic Bombs**: It is a malicious code written and secretly integrated into the operating system. As long as some specific information is fed into the system regularly, it does nothing to the system. Its malicious behaviour surfaces only when the required input is not fed into the system within a specified period. For example, assume that the owner of the code is an employee expecting to be fired and wants to disrupt the company's activities by destroying key records in the system. In such a case, the code is activated if it does not receive the necessary input from its owner for two consecutive working days. The related piece of code called the *time bomb* is activated on preset dates.

- **Worms:** Worms are malicious programs similar to viruses. Worms are typically standalone programs—therefore, they can execute without the help of host programs, and primarily replicate on networks. Some programs, which require user interaction to execute, may qualify to be worms if its main carrier is the network. The payload is designed to collect information from other machines in the network of current machine and construct an injection vector for those machines for which the information was collected. Then it starts its attack from there and continues.

- **Spyware:** These programs are designed with the purpose of monitoring browser habits of a user and to report that information back to the attacker whenever the target system invokes a browser. It can collect personal information stored in the system, extract internet surfing habits, etc. When a user requests a page from a Web server first time, it prepares a unique identification text for the future references of that user and it sends the text to the browser to store locally and use that identification for all its future web accesses. This information is called a *cookie*. The main purpose of cookies is to serve clients in a customized way. Usually, web servers maintain the information about its clients' web accesses and cookies are the keys to access such information. Spyware programs can use cookies to obtain user information. Once a spyware is set up in a system, it can install additional softwares, gain control of the browser, change settings of the system, etc. As online shopping is becoming common practice and most online shopping is done through web browsers, spyware programs can acquire critical information such as credit card numbers, bank account numbers, etc., and are, therefore, very dangerous.

≫ A spyware is a malicious program that enters a computer without permission and is capable of changing system configuration, monitoring internet activities, and sending information to the attacker.

To deal with crackers, we must first understand their nature. As we have seen, security attacks can come from both within the system as well as outside it. In both cases, the attacker may be a legitimate user or may impersonate a legitimate user. Additionally, in the case of an outside attacker, the channels used may be legitimate or illegitimate. There are numerous types of computer attacks and they continue to grow. In this book, we look at some attacks that are most frequently employed.

≫ A subversive code is essentially an executable code. It needs space in the memory to reside, space in the disk to store its files, access to network and its protocols to communicate with the outside from the system, and access to the login process to enter the system. If these potential sites are secured properly by operating systems, many security attacks can be avoided.

13.9.2 Attack Types

The types of attacks on computers are virtually limitless. We look at some of the more common types.

Trial and Error

The most obvious attempt to gather information is by random guesses. The most crucial information is the password and is the primary target of most

crackers. When someone wants to break into a computer, the first tactic is to guess the passwords of some legitimate users. Although the password domain is theoretically large, for individual users it is often small and, hence, makes random guess a useful effort. If the tastes, interests, and habits of a person are known, the likelihood of guessing the password of the person correctly is high. In addition, the random guess may be automated using a simple program. When the password domain is small, current high-speed computing could find the password in a reasonable length of time in some systems.

In a computer system, a cracker can attack a hardware element (e.g., the memory, CPU, Network) or a software element (data and programs). The attack could be an interruption (e.g., denial of service), interception (e.g., secretly stealing information), or modification (affecting integrity).

Spoofing

Spoofing is the way a cracker impersonates a legitimate user, a process (service), or a system, and tricks the user or application into revealing information. In these types of activities, popularly called *social engineering*, the attacker pretends to be a trusted user and convinces the target to reveal confidential information. It is the art of deception and such acts are sometimes called *masquerading*. In other words, it is a way to get hold of the identification of a legitimate user, a network host, or application and thereby gain an illegitimate advantage.

Another kind of spoofing and most of us are familiar with is e-mail requests. We often receive e-mails asking us to update our personal information relating to financial institutions or memberships. The motives of such mails from these attackers are to collect information and they appear to be sent from a legitimate source. This type of spoofing is called *phishing*. Information collected is then used to impersonate the original user. There are many types of spoofing. Some of the better-known spoofing techniques are:

- **Login Spoofing**: Most users enter the system by typing their login name and the password in an interface displayed locally on a screen or remotely through a web page. An attacker can write an interface that mimics the original interface. When a legitimate user enters the information, it collects the input, and then either logs in the user through genuine interface or simply asks the user to re-enter the information stating that the information already entered is incorrect, and disappears. Login spoofing may be considered a type of Trojan horse program.
- **IP Spoofing**: Internet Protocol (IP) is the protocol used at the network layer to transmit messages over the Internet. IP spoofing is the act of manipulating the header in the message to mask the cracker's true identity so that the message could appear as though it is from a trusted source. It essentially manipulates the source address field. When the receiver establishes the connection after the cracker forges the sender's address, many types of attacks are possible. For example, the cracker may
    - flood messages in the channel to overload the system. This might result in denial of service to regular users and may even lead to shutting down the system.

- impersonate between sender and receiver, and collect their messages by sending her own message on their behalf. That is, a message sent to the receiver process may be manipulated by the intruding cracker and re-sent as its own message. This is called *man-in-the-middle attack*.
- freely access data and load her agent programs.

IP spoofing can be prevented by monitoring the source and destination addresses and filtering the message if the address is not from a valid domain.

- **URL spoofing:** This is to display a false website address. The URL displayed is not really that of the site. For example, the user is asked to click a URL to go to the intended site. When the user clicks the link, the cracker extracts information such as the user's password and login id and modifies it before submitting it to the real webpage.
- **Web Spoofing:** This is making a false website appear to be another real website. There are many ways to do it. One of the simpler ways is to create an identical webpage and induce users to access the spoofed page believing that they are accessing the real webpage. A slightly more sophisticated way of web spoofing is becoming the middleman between the victim and the webpage and acting like the man-in-the-middle attack case. The users are tricked into believing they are working in a safe environment.
- **E-mail Spoofing:** It is to alter the e-mail header so that it appears to have originated from a familiar source when it is actually from a different source. E-mails are spoofed in many ways that can mislead their recipients. They may cause confusion or discredit a person, or sometimes even destroy reputations. It is mostly used for spamming—sending bulk messages to unsolicited recipients. E-mail spoofing is an interesting example for *social engineering*—the art of manipulating people to collect confidential information. For example, a cracker impersonates a system administrator, sends mails for password verification and then redirects the reply mails with their sensitive information to the cracker's own e-mail address.

Solutions to these attacks are problematic and beyond the scope of this book. Among the malicious codes, viruses and worms are generic and highly pervasive. They have many common characteristics. Next, we look at the types of viruses, and how they operate.

13.10 Virus

Computer virus is a very familiar term for most people in that it sounds something similar to real virus and that it infects computers. Most viruses are

designed for manifold disruptive purposes. A virus could be innocuous, humorous, data altering, or even catastrophic. Mainly, it is a program and does what its creator asks it to do. The replicating nature of most viruses makes them so powerful that they can easily multiply to create many undesirable states. There are many types of viruses based on their infection strategies. We describe some common types of viruses.

- *Boot Virus*: In the early PCs, the virus used the bootstrap procedure. Initially, the ROM-BIOS loaded the first sector of the boot-disk into the main memory. Then the loaded program executed to bring up the remaining part of operating system. The code that brings up the rest of the operating system is called the bootstrap program. If the virus code replaces the bootstrap program at the first sector of the boot disk, then the virus acquires control of the system as soon as it is loaded. These viruses typically hook the disk interrupt handler routine and start to monitor for diskette accesses. The hooking is done by moving the original interrupt routine to a different location and occupying its place. When the disk interrupt occurs, the control passes to the virus instead of the actual interrupt handler routine. It then uses the actual interrupt routine to its advantage. When a diskette is accessed, its copy is written to the first sector of the diskette. This way the virus infects the diskettes used in the computer. Since the diskette method of booting is outdated, such boot viruses do not exist in modern systems.

- *File virus*: File viruses locate one or more files in the disk and write their intended code or information into them. There are many forms the files could be written. The simplest form is to overwrite the files with its own copy. The virus can search for a particular file type, for example, executable files, and write its own code. It can write anywhere and in anyway: at the top, at the bottom, top and bottom, random insertion with appropriate jumps, etc. Different techniques may be used to make cure difficult. These viruses are executed only when the file is executed and, therefore, they usually spread at a much slower speed.

- *Memory-Resident Virus*: These viruses remain in the memory after they are installed and initiated. Typically, they get the control of the system, allocate a block of memory for their code, load their code into the allocated block, and starts execution to infect the files or programs in the system. Some viruses load themselves directly into specified memory locations without requiring memory allocations. Most viruses written for DOS operating systems typically hooked into interrupt routines and then gain control whenever such interrupt occurs.

- *Viruses in Processes*: These viruses are common in multitasking operating systems. Usually processes are protected using address spaces. The virus stays in the address space of that process and can affect only the data in that space if it is a user process. Some viruses

≫ A ninth-grade student Rick Skrenta wrote the first true virus "Elk Cloner" in 1982 for Apple II. Rick did not think that the program would work well, but his friend found it entertaining and eventually got it to infect his math teacher's computer. The payload printed Rick's poem on the screen when reset was pressed after the 50th reboot of the system. A counter updated every reboot of the system that was infected with the payload, and invoked its own reset handler to display the poem when the counter value reached 50.

≫ The first widely known successful computer virus, called Brain, was written in 1986 on the IBM PC. In those days, the machines would boot from floppy diskettes and that provided a great opportunity to load the virus before the operating system gets control of the system. (A floppy diskette is magnetic data storage medium that was in use between the 1970s and 1990s.)

could get into kernel address space by inserting itself into device drivers. Device drivers are typically executed in the kernel mode and, therefore, a virus stays in the kernel mode. These viruses are more dangerous than user-process viruses.

Viruses, once entered into the system, can be activated in many ways: based on events (e.g., interrupts, timer, date, etc.), or whenever the host process is invoked. Viruses that can operate in the kernel mode can access and change anything in the system. Device-driver viruses infect the device drivers and operate in kernel mode. A virus typically has two logical components: (a) *payload* to do the intended tasks and (b) a *dropper* or injection vector to insert the payload in suitable places in the program or memory. In generic terms, we use dropper as way to spread the virus.

≫ A virus simply exploits the potential weaknesses of a particular environment. A virus designed for one environment need not work in another environment. A virus can successfully penetrate a system only if its requirements (referred to as dependencies) match the environment in the system. It could be architecture-dependent, particular CPU-dependent, operating system-dependent, file system-dependent, file format-dependent, language-dependent, etc.

13.10.1 Droppers

Droppers are installers for first-generation virus codes. The purpose of a dropper is to place the payload and set it up for execution. In the early days, when diskettes were used as the primary communication mechanism, viruses were spread through diskettes. In modern days, as computers are connected to the Internet, viruses are spread through the latter. For example, an attacker can write a virus, insert it into attractive applications (such as games, screen savers, etc.) or tools, and put it in a shareware web site. People interested in the program download it and start executing it without knowing that they invited the virus into the system.

13.10.2 Payloads

A simplest payload program for earlier UNIX systems is to call the *fork()* system call within an infinite loop. This will quickly fill the process table leading to denial of service. To prevent this, modern UNIX systems have a limit on the number of children a process can have at a time. Literally, there is no limit for the way the payloads can be designed. Some common payload types are as follows.

- *Non-destructive Payload*: These types of viruses do not do any harm to the system except displaying messages on the screen.
- *Replication-only Payload*: Many viruses are not intended to harm the system except replicating itself. However, the replication may accidentally cause destruction such as overwriting a file or part of the disk or memory.
- Somewhat Destructive Payload: These viruses usually target executable files such as antivirus software, firewall programs, etc.
- *Highly Destructive Payload*: These are the most dangerous group of viruses designed intentionally for harming software or hardware.

They could infect executables, overwrite data, slowly manipulate data (*data diddlers*), steal data, destroy hardware, etc. Most modern systems use Flash BIOS for easy update. These updates could be done by software. There are viruses designed to attack Flash BIOS and are very dangerous.

13.10.3 Detection and Counter Measures

Detection of malicious attacks or infection of the system is not always easy. We list some of the common symptoms that signal the system is infected.

- Change in file size or date of modification or creation
- Slow starting of the system
- Slow speed
- Unexpected or frequent system failures or restarts
- Low system memory or disk space
- Unable to start particular applications

These and any unusual behaviour of the system are indications that the system may be under attack. After it is determined or even suspected that the system is infected by a malicious code, virus needs to be located and removed from the system. This is the task of anti-virus software. They scan the system to identify the infected locations and then take suitable action to cure the system. The scanning involves looking for unusual patterns not common in regular files. If this unusual pattern typically belongs to a particular virus, then the pattern is known as the *signature* of that virus. Once a signature is found, then a virus-specific algorithm is applied to identify the exact virus and remove it from the system.

13.11 Application Security Approaches

Security software are designed either to block the suspected elements from entering the system or to fight (find and eliminate) the malicious programs that entered the system. For our discussion, we refer to them, respectively, as *blockers* and *fighters*. There is a third kind of software designed to learn the tactics and motives of attackers in order to design effective blockers and fighters. We refer to this type of software or approach as *trappers*. We look at some trappers, blockers, and fighters next.

13.11.1 Firewall and Pop-up Blockers

When a computer is connected to the Internet, it becomes one among several millions of computers already connected to it. To keep private information within the system private and the computer safe from the external attackers,

the elements (code and data) flowing in and out must be controlled appropriately. The IP address of the computer and port addresses within the computer are basic information used to design security policies to control such traffic. The IP address is the connection point of the computer to the Internet and the port addresses in the computer are the connection points to the processes and services within the computer. Firewalls use the information to block illegitimate traffic. Additionally, application level information such as user identifications, organization names, etc., are also used to design higher-level security policies indicating what can get in and what can get out of the system.

The firewall is the most popular software designed to block suspicious Internet traffic. It is usually installed in gateways and routers to filter out harmful data- and access requests. It can prevent worms and other attacks in a number of ways. Router level firewalls typically allow a specific IP address to use a specific port to access specified services. Any other combinations are simply denied. A simpler way is to block the use of any ports that need not be used and are also vulnerable to attacks.

Firewalls could be very generic or application (user) specific. Application specific firewalls can provide better security because they are based on more knowledge about the application context. Application specific firewalls are called *proxy firewalls*. Of course, firewalls can often be circumvented by suitable spoofing if the crackers know information such as the IP addresses, ports, services, etc., used by the firewall to filter accesses.

≫ The objective of firewalls is to make the systems invisible to the attackers. It denies the attackers entry to system based on a given security policy.

Pop-ups display unsolicited messages on the monitor with the intent of advertising or collecting e-mail addresses. JavaScript usually generates pop-ups. Pop-ups are indications that some unwanted elements have entered the system. Blocking pop-ups is a way of reducing the risk in the system. Most browsers offer security settings to block pop-ups.

13.11.2 Anti-virus and Anti-spam Software

≫ Currently there exist more than 50,000 viruses, and about 200 new viruses appear each month. A 2006 study indicates that 89% of computers were infected by spyware when the study was conducted.

The fundamental objective of anti-virus- and anti-spam software is to block-, detect-, and/or disinfect malicious programs. They use different filtering- and scanning techniques to detect and block illegitimate elements in the system. Disinfection involves carefully removing the malicious code from the system and repairing the damage. These use heavily automated reverse engineering techniques to analyze the malicious code. Some damages are irreversible. Known malicious codes are classified and their analysis and knowledge are saved for future use in defence. Such knowledge about known viruses is the basis for anti-virus software. In order to be up-to-date and effective, these software must be updated as soon as the information about the new virus is available

13.11.3 Sandboxing

Sandboxing is a way of providing controlled execution environment for entrusted programs. The idea is to contain the execution of a code in such a

way that it cannot cause any damage outside its confined area. A typical sandboxing involves placing restrictions on file accesses and limiting network connections. Sandboxing is semantically equivalent to protection domains discussed earlier. The scope of sandboxing could be a simple virtual machine or a complete operating system. A related concept is *virtualization*. Virtualization is creating an execution environment for various purposes such as safe execution, convenience, quality of service, portability, etc. Examples of virtualization include Java virtual machine, User Mode Linux (UML), Parallel Virtual Machine (PVM), Grid Computing, etc. In this sense, sandboxing is a virtualization technique for safe executions.

Sandboxing is widely used in Java providing a reserved area, the sandbox, for Java interpreter inside Internet browsers. This limits the interpreter to perform its execution only inside the sandbox and avoids access to the resources outside. This is implemented by assigning a security manager for each program/applet. The security policies are specified in a customizable file and the security manager enforces the policies during the execution of the code inside sandbox.

13.11.4 Software Updates/Patching

Patching is a way of repairing an executable program, adding a new feature, or fixing a bug. A patch is a piece of object code usually developed and distributed as a replacement for or an insertion in executable program in the system. As it is hard to design software systems without vulnerabilities, patching is an effective approach practised to enhance security of the system. That is, whenever a weakness is identified in the system a solution is designed and patched to the system to fix it.

13.11.5 Honeypot Systems

Honeypot systems are decoy systems designed to attract attackers in order to learn their tactics and motives. They can be emulation systems or real systems. When a worm enters the honeypot, it is captured and sent for analysis. It is a trapper program. NetCat tool is an example of honeypot. It reads- and writes data across network connections. For example, consider the following command.

```
NC -l -p 80 > listen80.log
```

This command instructs NetCat to listen on port 80 and redirect the incoming traffic to the file listen80.log. The log file is then analyzed to find any illegal traffic.

In essence, as the computer system is vulnerable to attack from many directions, so there is no single approach to assure security. A combination of strategies such as suitably controlling access to system resources, virtualization, sandboxing, namespace management, partitioning, using firewalls, blockers, and anti-virus and anti-spam software, can be used to provide application security to computer systems.