Module 5

Operating System Security: Memory and Address Protection, Control of Access to General Objects, File Protection Mechanisms, Models of Security – Bell-La Padula Confidentiality Model and Biba Integrity Model.

System Security: Intruders, Intrusion Detection, Password Management, Viruses and Related Threats, Virus Countermeasure.

Operating System Security

The protection mechanism is the mechanism used by specific operating system to safeguard information in a computer system. Protection features provided by general-purpose operating systems includes protecting memory, files, and the execution environment, Controlled access to objects and User authentication.

Memory and Address Protection

The most obvious problem of multiprogramming is preventing one program from affecting the data and programs in the memory space of other users. Protection can be built into the hardware mechanisms that control efficient use of memory, so solid protection can be provided at essentially no additional cost.

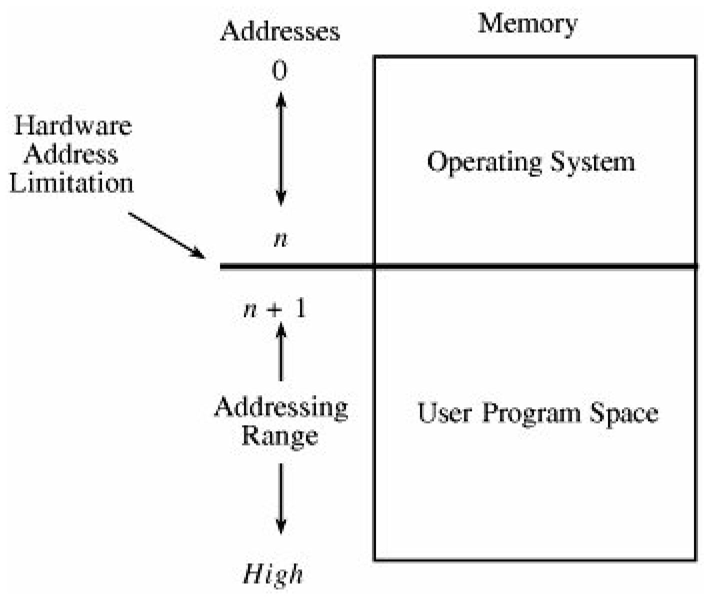
Fence: - The simplest form of memory protection was introduced in single-user operating systems to prevent a faulty user program from destroying part of the resident portion of the operating system. A fence is a method to confine users to one side of a boundary.

Mainly fence is implemented in two ways

- Fixed Fence
- Fence register :-Variable Fence Register

Below is the figure showing a fixed fence. The fence was a predefined memory address, enabling the operating system to reside on one side and the user to stay on the other. This kind of implementation was very restrictive because a predefined amount of space was always reserved for the operating system, whether it was needed or not. If less than the predefined space was required, the excess space was wasted. Conversely, if the operating system needed more space, it could not grow beyond the fence boundary.

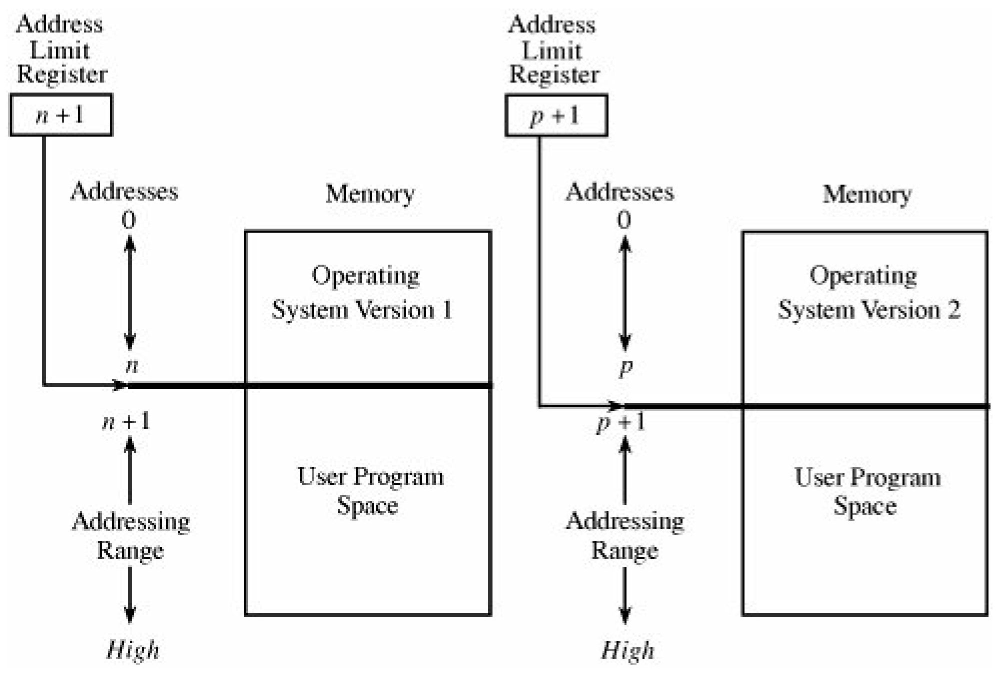
Figure 1. Fixed Fence.



Another implementation used a hardware register, often called a **fence register**, containing the address of the end of the operating system. In contrast to a fixed fence, in this scheme the location of the fence could be changed. Each time a user program generated an address for data modification, the address was automatically compared with the fence address. If the address was greater than the fence address (that is, in the user area), the instruction was executed; if it was less than the fence address (that is, in the operating system area), an error condition was raised. The use of fence registers is shown below.

A fence register protects only in one direction. In other words, an operating system can be protected from a single user, but the fence cannot protect one user from another user. A user cannot identify certain areas of the program as inviolable (such as the code of the program itself or a read-only data area).

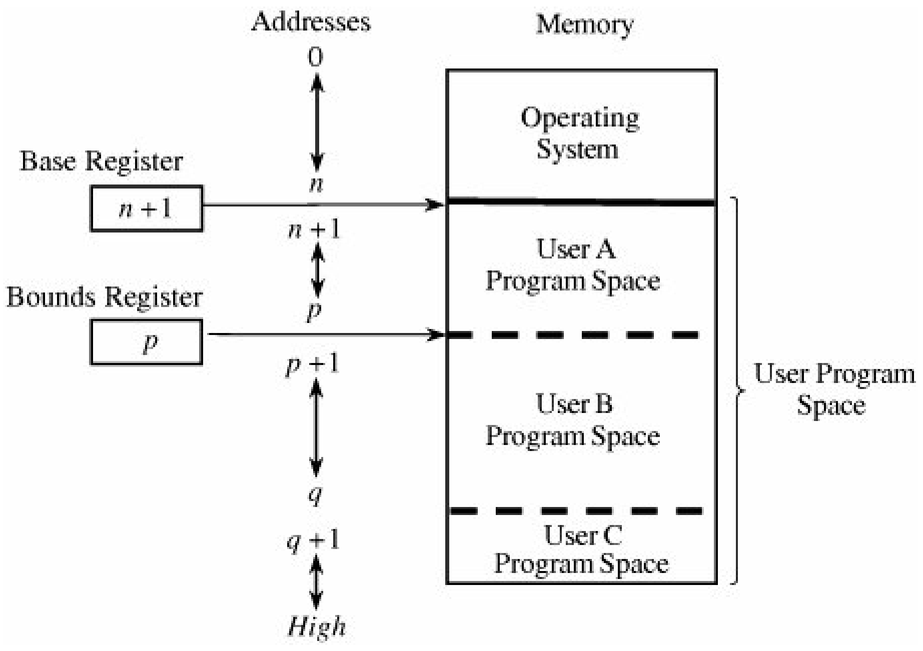
Figure 2. Variable Fence Register.



Relocation: - It is the process of taking a program written as if it began at address 0 and changing all addresses to reflect the actual address at which the program is located in memory, adding a constant relocation factor to each address of the program. That is, the relocation factor is the starting address of the memory assigned for the program. The fence register can be a hardware relocation device. The contents of the fence register are added to each program address. This action both relocates the address and guarantees that no one can access a location lower than the fence address.

Base/Bounds Registers: - A variable fence register is generally known as a base register. Fence registers provide a lower bound (a starting address) but not an upper one. An upper bound can be useful in knowing how much space is allotted and in checking for overflows into "forbidden" areas. To overcome this difficulty, a second register called a bounds register, is an upper address limit, in the same way that a base or fence register is a lower address limit. Each program address is forced to be above the base address because the contents of the base register are added to the address; each address is also checked to ensure that it is below the bounds address. In this way, a program's addresses are neatly confined to the space between the base and the bounds registers.

Figure 3. Pair of Base/Bounds Registers.

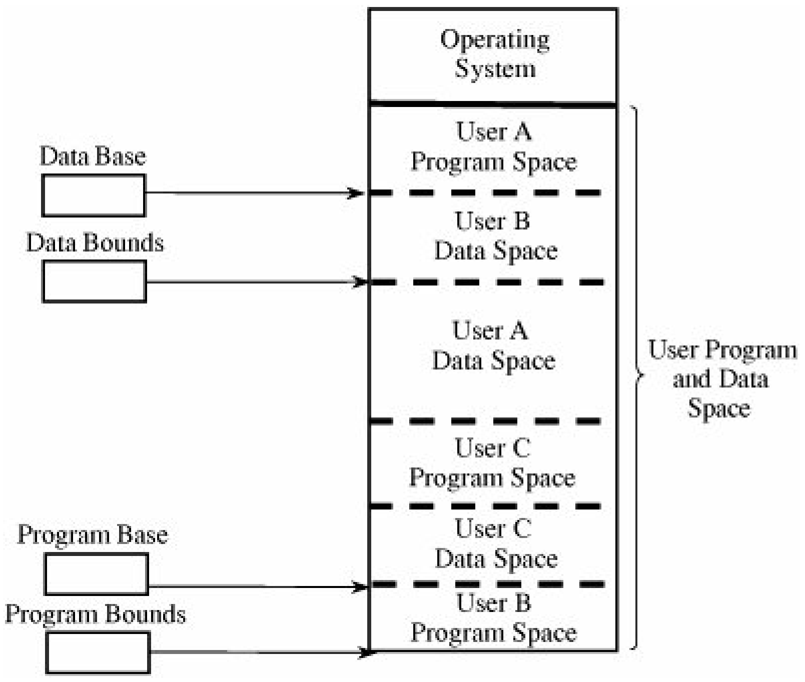


This technique protects a program's addresses from modification by another user. When execution changes from one user's program to another's, the operating system must change the contents of the base and a bound registers to reflect the true address space for that user. This change is part of the general preparation, called a context switch that the operating system must perform when transferring control from one user to another.

With a pair of base/bounds registers, a user is perfectly protected from outside users, or, more correctly, outside users are protected from errors in any other user's program. Erroneous addresses inside a user's address space can still affect that program because the base/bounds checking guarantees only that each address is inside the user's address space. For example, a user error might occur when a subscript is out of range or an undefined variable generates an address reference within the user's space but, unfortunately, inside the executable instructions of the user's program. In this manner, a user can accidentally store data on top of instructions. Such an error can let a user inadvertently destroy a program, but (fortunately) only the user's own program.

The overwriting problem can be avoided by using another pair of base/bounds registers, one for the instructions (code) of the program and a second for the data space. Then, only instruction fetches (instructions to be executed) are relocated and checked with the first register pair, and only data accesses (operands of instructions) are relocated and checked with the second register pair. Although two pairs of registers do not prevent all program errors, they limit the effect of data-manipulating instructions to the data space. The pairs of registers offer another more important advantage: the ability to split a program into two pieces that can be relocated separately.

Figure 4. Two Pairs of Base/Bounds Registers.



These two features seem to call for the use of three or more pairs of registers: one for code, one for read-only data, and one for modifiable data values. Although in theory this concept can be extended, two pairs of registers are the limit for practical computer design. For each additional pair of registers (beyond two), something in the machine code of each instruction must indicate which relocation pair is to be used to address the instruction's operands. That is, with more than two pairs, each instruction specifies one of two or more data spaces. But with only two pairs, the decision can be automatic: instructions with one pair, data with the other.

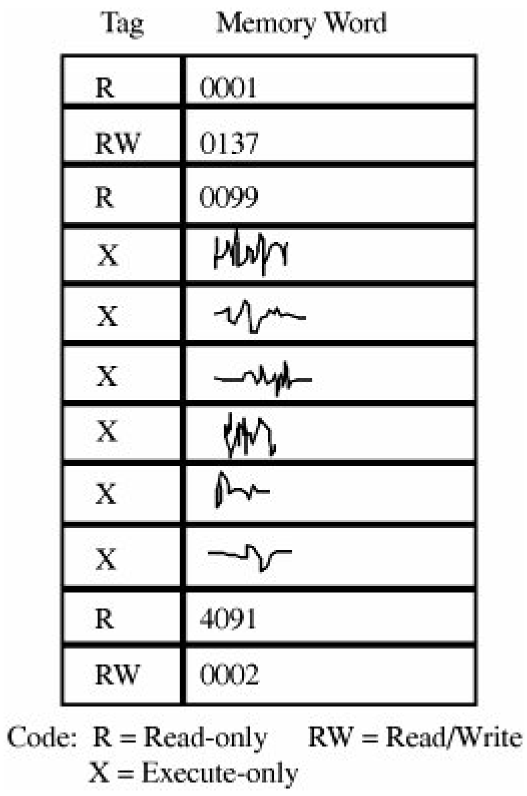
Another problem with using base/bounds registers for protection or relocation is their contiguous nature. Each pair of registers confines accesses to a consecutive range of addresses. A compiler or loader can easily rearrange a program so that all code sections are adjacent and all data sections are adjacent.

Integrity of certain data values by allowing them to be written when the program is initialized but prohibiting the program from modifying them later. A programmer may also want to invoke a shared subprogram from a common library. We can address some of these issues by using good design. These characteristics dictate that one program module must share with another module only the minimum amount of data necessary for both of them to do their work.

Base/bounds registers create an all-or-nothing situation for sharing. Either a program makes all its data available to be accessed and modified or it prohibits access to all. Even if there were a third set of registers for shared data, all data would need to be located together. A procedure could not effectively share data items A, B, and C with one module, A, C, and D with a second, and A, B, and D with a third. The only way to accomplish the kind of sharing we want would be to move each appropriate set of data values to some contiguous space. However, this solution would not be acceptable if the data items were large records, arrays, or structures.

Tagged Architecture: -Every word of machine memory has one or more extra bits to identify the access rights to that word. These access bits can be set only by privileged (operating system) instructions. The bits are tested every time an instruction accesses that location.

Figure 5. Example of Tagged Architecture.



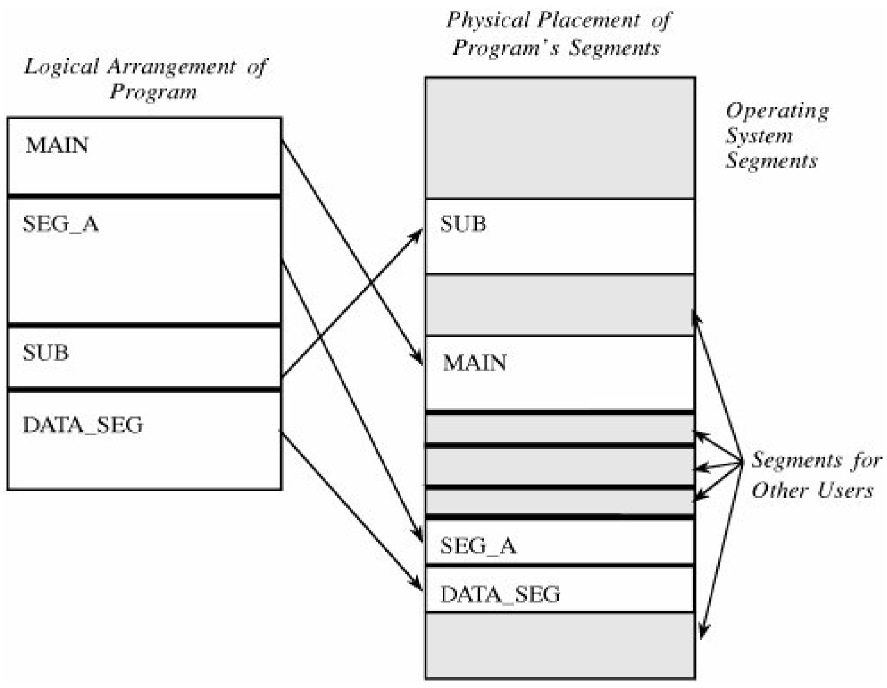
Tag	Memory Word
R	0001
RW	0137
R	0099
X	
X	
X	
X	
X	
X	
R	4091
RW	0002

Code: R = Read-only RW = Read/Write
X = Execute-only

A variation used one tag that applied to a group of consecutive locations, such as 128 or 256 bytes. With one tag for a block of addresses, the added cost for implementing tags was not as high as with one tag per location. A tagged architecture would require fundamental changes to substantially all the operating system code, a requirement that can be prohibitively expensive. But as the price of memory continues to fall, the implementation of a tagged architecture becomes more feasible.

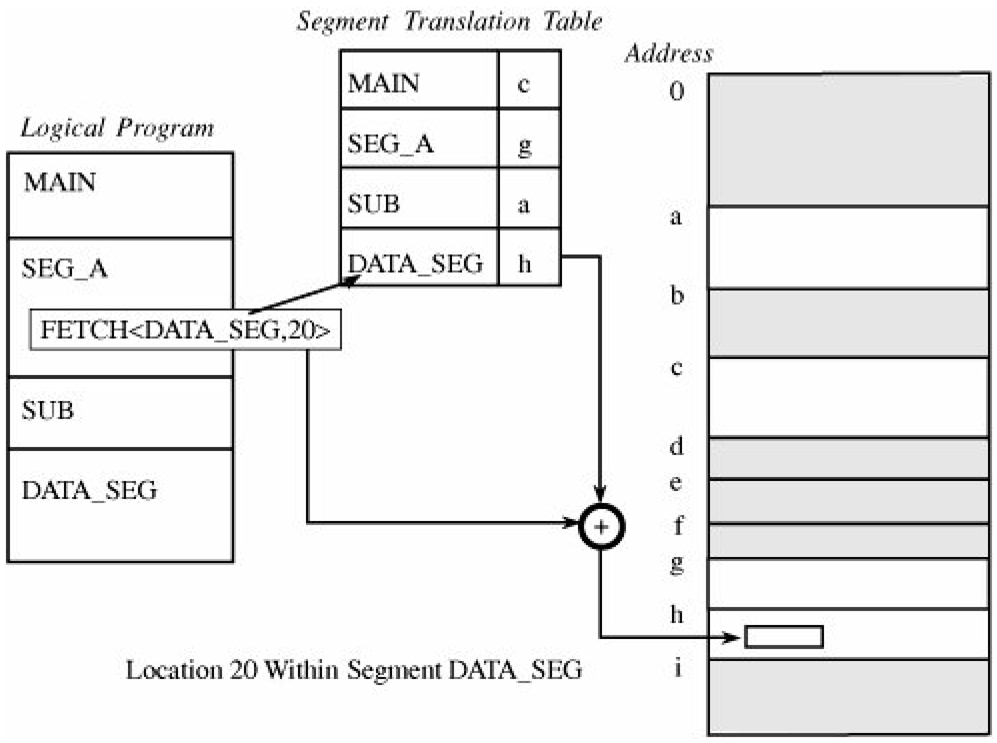
Segmentation: - It involves the simple notion of dividing a program into separate pieces. Each piece has a logical unity, exhibiting a relationship among all of its code or data values. Segmentation was developed as a feasible means to produce the effect of the equivalent of an unbounded number of base/bounds registers. Segmentation allows a program to be divided into many pieces having different access rights. Each segment has a unique name. A code or data item within a segment is addressed as the pair <name, offset>, where name is the name of the segment containing the data item and offset is its location within the segment (that is, its distance from the start of the segment). Logically, the programmer pictures a program as a long collection of segments. Segments can be separately relocated, allowing any segment to be placed in any available memory locations.

Figure 6. Logical and Physical Representation of Segments.


Logical Arrangement of Program
MAIN
SEG_A
SUB
DATA_SEG
Physical Placement of Program's Segments
Operating System Segments
SUB
MAIN
SEG_A
DATA_SEG
Segments for Other Users

The operating system must maintain a table of segment names and their true addresses in memory. When a program generates an address of the form <name, offset>, the operating system looks up name in the segment directory and determines its real beginning memory address. To that address the operating system adds offset, giving the true memory address of the code or data item. Two processes that need to share access to a single segment would have the same segment name and address in their segment tables.

Figure 7. Translation of Segment Address.



A user's program does not know what true memory addresses it uses. The <name, offset> pair is adequate to access any data or instruction to which a program should have access. This hiding of addresses has three advantages for the operating system.

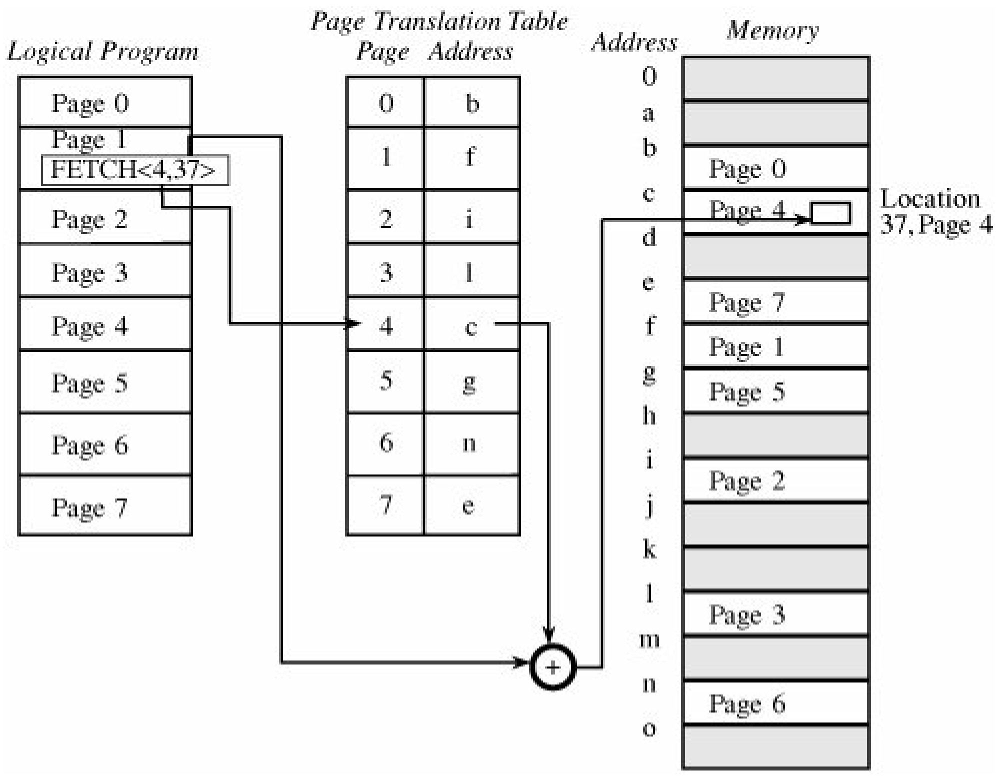
1. The operating system can place any segment at any location or move any segment to any location, even after the program begins to execute. Because it translates all address references by a segment address table, the operating system needs only update the address in that one table when a segment is moved.

2. A segment can be removed from main memory (and stored on an auxiliary device) if it is not being used currently.
3. Every address reference passes through the operating system, so there is an opportunity to check each one for protection.

A process can access a segment only if that segment appears in that process's segment translation table. The operating system controls which programs have entries for a particular segment in their segment address tables. This control provides strong protection of segments from access by unpermitted processes. Segmentation offers these security benefits:

1. Each address reference is checked for protection.
2. Many different classes of data items can be assigned different levels of protection.
3. Two or more users can share access to a segment, with potentially different access rights.
4. A user cannot generate an address or access to an unpermitted segment.

Figure 8. Page Address Translation.



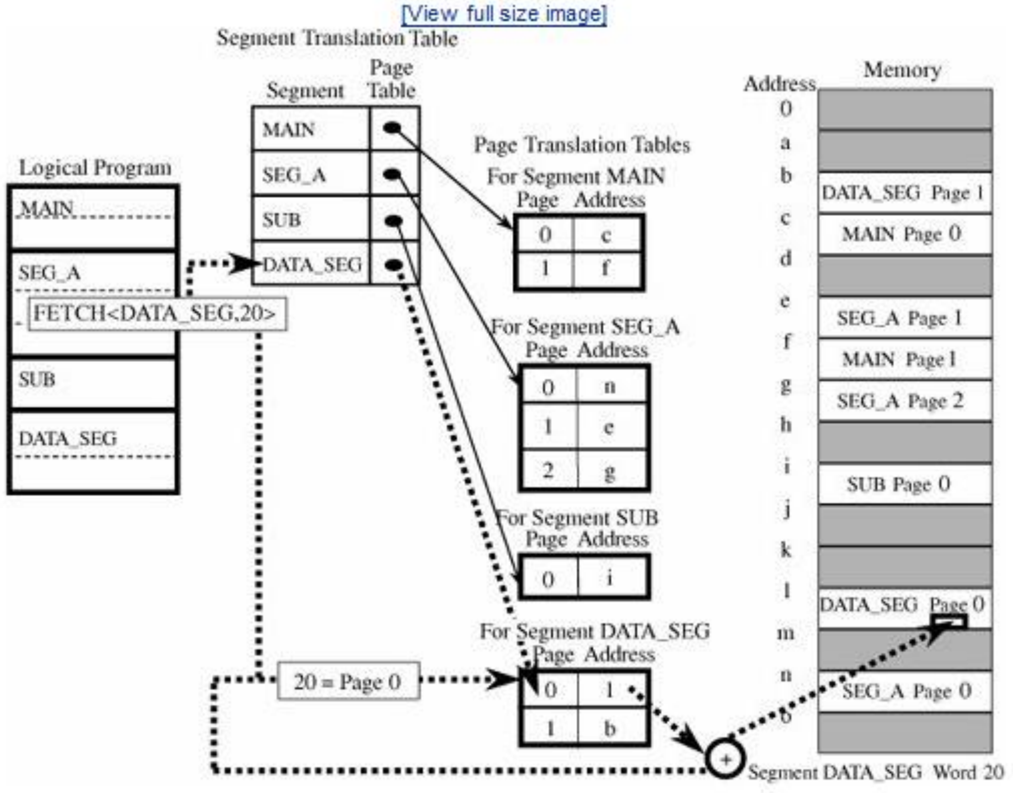
Paging: - The program is divided into equal-sized pieces called pages, and memory is divided into equal-sized units called **page frames**. As with segmentation, each address in a paging scheme is a two-part object, this time consisting of <page, offset>. Each address is again translated by a process similar to that of segmentation. The operating system maintains a table of user page numbers and their true addresses in memory. The page portion of every <page, offset> reference is converted to a page frame address by a table lookup; the offset portion is added to the page frame address to produce the real memory address of the object referred to as <page, offset>.

The paging offers implementation efficiency, while segmentation offers logical protection characteristics.

Combined Paging with Segmentation: - Paging offers implementation efficiency, while segmentation offers logical protection characteristics. Since each approach has drawbacks as well as desirable features, the two approaches have been combined.

Figure 9. Paged Segmentation.

[View full size image]



.

Control of Access to General Objects

Here is the example of the kinds of objects for which protection is desirable:

1. memory
2. a file or data set on an auxiliary storage device
3. an executing program in memory
4. a directory of files
5. a hardware device
6. a data structure, such as a stack
7. a table of the operating system
8. instructions, especially privileged instructions
9. passwords and the user authentication mechanism
10. the protection mechanism itself

All accesses to memory occur through a program, which is referred to be a program or the programmer as the accessing agent. There are several complementary goals in protecting objects.

- Check every access: - There are the cases where we may want to revoke a user's privilege to access an object. If we have previously authorized the user to access the object, we do not necessarily intend that the user should retain indefinite access to the object. In some situations, we may want to prevent further access immediately after we revoke authorization. For this reason, every access by a user to an object should be checked.
- Enforce least privilege: - The principle of least privilege states that a subject should have access to the smallest number of objects necessary to perform some task. Even if extra information would be useless or harmless if the subject were to have access, the subject should not have that additional access. For example, a program should not have access to the absolute memory address to which a page number reference translates, even though the program could not use that address in any effective way. Not allowing access to unnecessary objects guards against security weaknesses if a part of the protection mechanism should fail.
- Verify acceptable usage: - Ability to access is a yes-or-no decision. But it is equally important to check that the activity to be performed on an object is appropriate. For

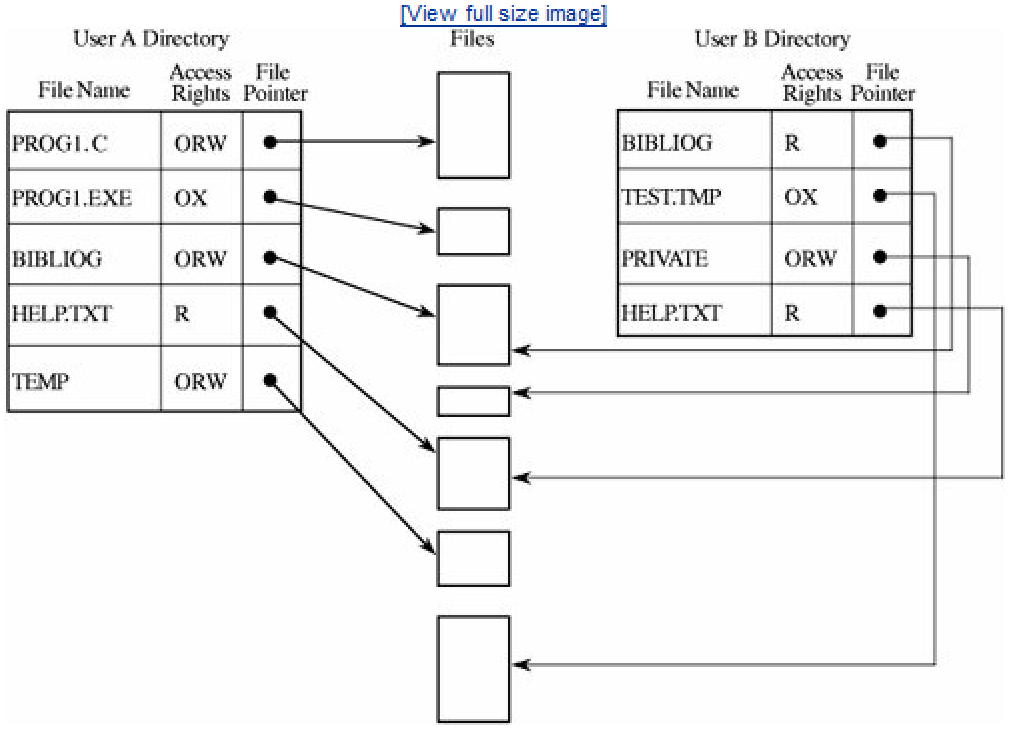
example, a data structure such as a stack has certain acceptable operations, including push, pop, clear, and so on. We may want not only to control who or what has access to a stack but also to be assured that the accesses performed are legitimate stack accesses.

Directory

One simple way to protect an object is to use a mechanism that works like a file directory. Imagine we are trying to protect files (the set of objects) from users of a computing system (the set of subjects). Every file has a unique owner who possesses "control" access rights (including the rights to declare who has what access) and to revoke access to any person at any time. Each user has a file directory, which lists all the files to which that user has access. No user can be allowed to write in the file directory because that would be a way to forge access to a file. Therefore, the operating system must maintain all file directories, under commands from the owners of files. The obvious rights to files are the common read, write, and execute familiar on many shared systems. Furthermore, another right, owner, is possessed by the owner, permitting that user to grant and revoke access rights. Below figure shows an example of a file directory.

Figure 10. Directory Access.

[View full size image]



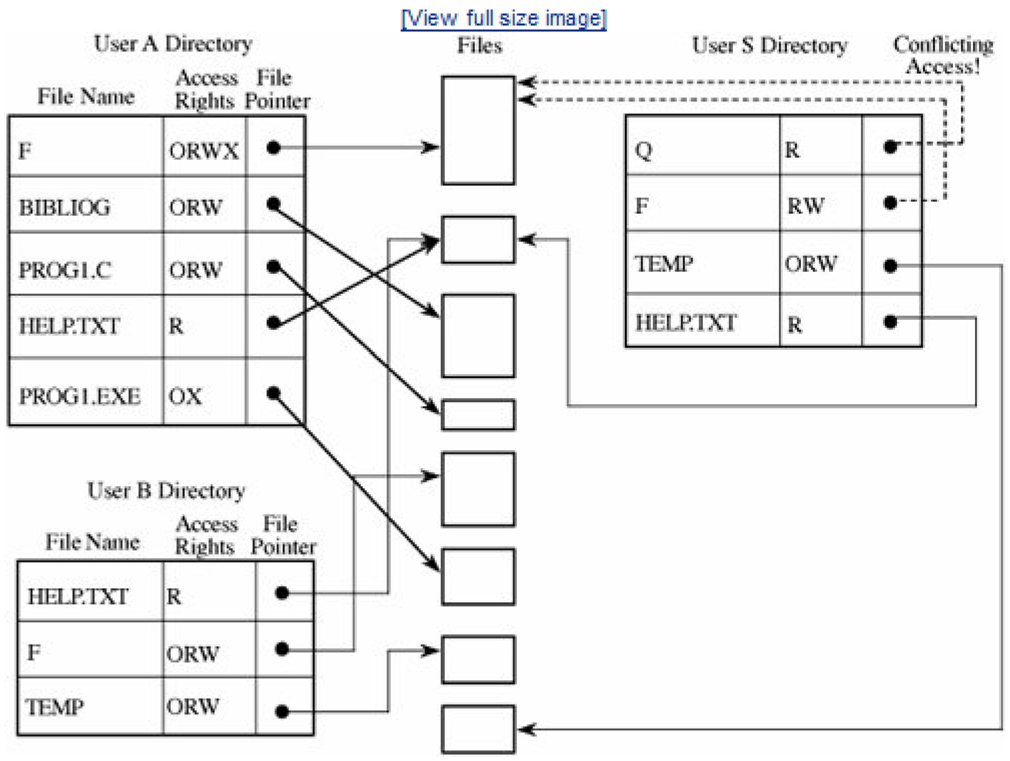
This approach is easy to implement because it uses one list per user, naming all the objects that user is allowed to access. But there are several problems in this method. First, the list becomes too large if many shared objects, such as libraries of subprograms or a common table of users, are accessible to all users. The directory of each user must have one entry for each such shared object, even if the user has no intention of accessing the object. Deletion must be reflected in all directories.

A second difficulty is **revocation of access**. If owner A has passed to user B the right to read file F, an entry for F is made in the directory for B. This granting of access implies a level of trust between A and B. If A later questions that trust, A may want to revoke the access right of B. The operating system can respond easily to the single request to delete the right of B to access F because that action involves deleting one entry from a specific directory. But if A wants to remove the rights of everyone to access F, the operating system must search each individual directory for the entry F, an activity that can be time consuming on a large system. B may have passed the access right for F to another user, so A may not know that F's access exists and should be revoked. This problem is particularly serious in a network.

Owners A and B may have two different files named F, and they may both want to allow access by S. Clearly, the directory for S cannot contain two entries under the same name for different files. Therefore, S has to be able to uniquely identify the F for A (or B). One approach is to include the original owner's designation as if it were part of the file name, with a notation such as A:F (or B:F).

S has trouble remembering file contents from the name F. Another approach is to allow S to name F with any name unique to the directory of S. Then, F from A could be called Q to S. As shown in below figure, S may have forgotten that Q is F from A, and so S requests access again from A for F. But by now A may have more trust in S, so A transfers F with greater rights than before. This action opens up the possibility that one subject, S, may have two distinct sets of access rights to F, one under the name Q and one under the name F. In this way, allowing pseudonyms leads to multiple permissions that are not necessarily consistent. Thus, the directory approach is probably too simple for most object protection situations.

Figure 11. Alternative Access Paths.

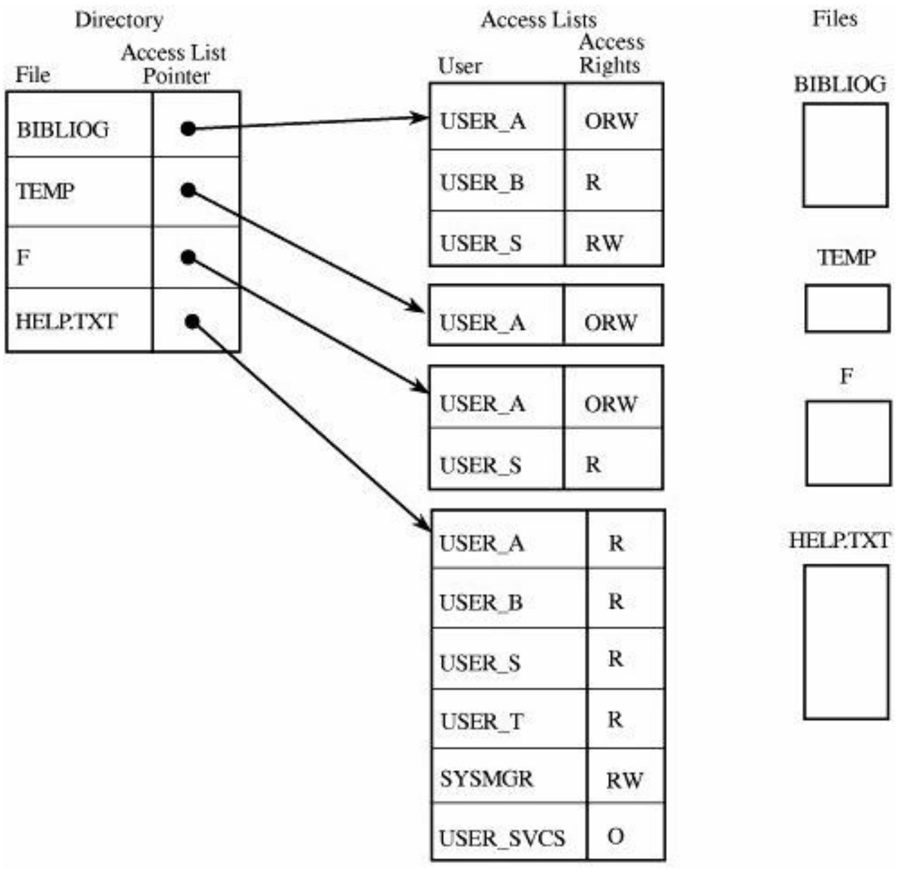


Access Control List

In access control list there is one such list for each object, and the list shows all subjects who should have access to the object and what their access is. This approach differs from the directory list because there is one access control list per object; a directory is created for each subject. Although this difference seems small, there are some significant advantages.

Consider subjects A and S, both of whom have access to object F. The operating system will maintain just one access list for F, showing the access rights for A and S, as shown in below figure. The access control list can include general default entries for any users. In this way, specific users can have explicit rights, and all other users can have a default set of rights. With this organization, a public file or program can be shared by all possible users of the system without the need for an entry for the object in the individual directory of each user.

Figure 12. Access Control List.



The Multics OS used a form of access control list in which each user belonged to three protection classes: a user, a group, and a compartment. The user designation identified a specific subject, and the group designation brought together subjects who had a common interest, such as coworkers on a project. The compartment confined an untrusted object; a program executing in one compartment could not access objects in another compartment without specific permission. The compartment was also a way to collect objects that were related, such as all files for a single project.

If Adams logs in as user Adams in group Decl and compartment Art2, only objects having Adams-Decl-Art2 in the access control list are accessible in the session. The solution is the use of wild cards, meaning placeholders that designate "any user" (or "any group" or "any compartment"). An access control list might specify access by Adams-Decl-Art1, giving specific rights to Adams if working in group Decl on compartment Art1. The list might also specify

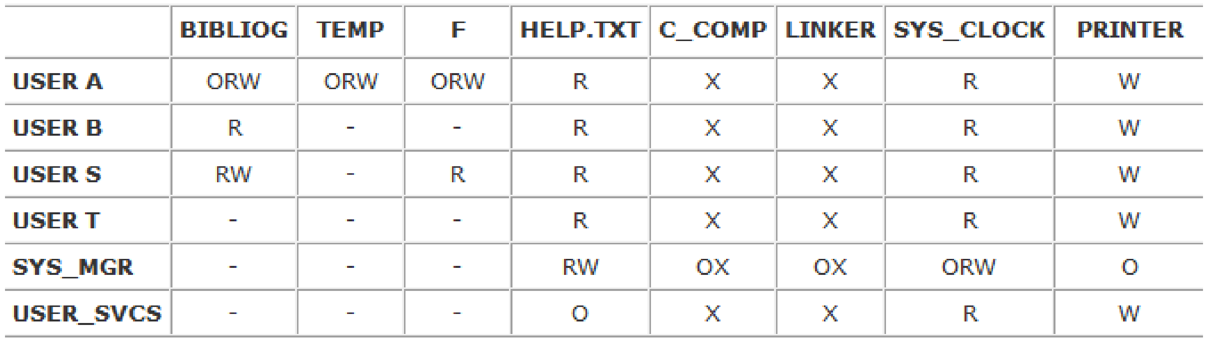
Adams-*-Art1, meaning that Adams can access the object from any group in compartment Art1. *-Decl-* would mean "any user in group Decl in any compartment."

The access control list can be maintained in sorted order, with * sorted as coming after all specific names. For example, Adams-Decl-* would come after all specific compartment designations for Adams. The search for access permission continues just until the first match. The last entry on an access list could be *-*-*, specifying rights allowable to any user not explicitly on the access list.

Access Control Matrix

In access control matrix, a table in which each row represents a subject, each column represents an object, and each entry is the set of access rights for that subject to that object. An example representation of an access control matrix is shown in below table. In general, the access control matrix is sparse (meaning that most cells are empty): Most subjects do not have access rights to most objects. The access matrix can be represented as a list of triples, having the form <subject, object, rights>. Searching a large number of these triples is inefficient enough that this implementation is seldom used.

Table 1. Access Control Matrix.



	BIBLIOG	TEMP	F	HELP.TXT	C_COMP	LINKER	SYS_CLOCK	PRINTER
USER A	ORW	ORW	ORW	R	X	X	R	W
USER B	R	-	-	R	X	X	R	W
USER S	RW	-	R	R	X	X	R	W
USER T	-	-	-	R	X	X	R	W
SYS_MGR	-	-	-	RW	OX	OX	ORW	O
USER_SVCS	-	-	-	O	X	X	R	W

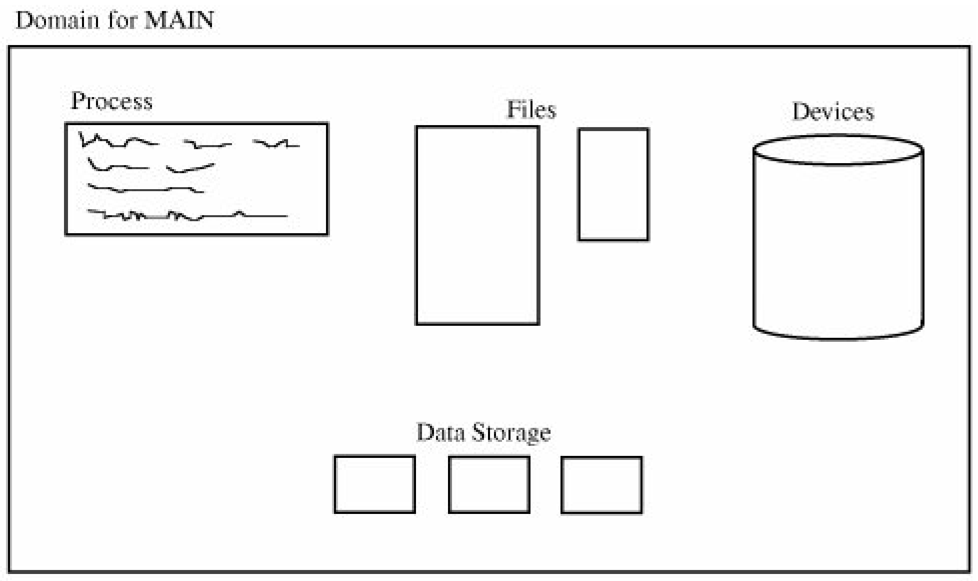
Capability

Capability is an unforgeable token that gives the possessor certain rights to an object. A capability is a ticket giving permission to a subject to have a certain type of access to an object. The ticket must be unforgeable. A capability can be created only by a specific request from a user to the operating system. Each capability also identifies the allowable accesses. Capabilities can be encrypted under a key available only to the access control mechanism. If the encrypted capability contains the identity of its rightful owner, user A cannot copy the capability and give it to user B.

One possible access right to an object is transfer or propagates. A subject having this right can pass copies of capabilities to other subjects. In turn, each of these capabilities also has a list of permitted types of accesses, one of which might also be transfer. In this instance, process A can pass a copy of a capability to B, who can then pass a copy to C. B can prevent further distribution of the capability (and therefore prevent further dissemination of the access right) by omitting the transfer right from the rights passed in the capability to C. B might still pass certain access rights to C, but not the right to propagate access rights to other subjects.

As a process executes, it operates in a **domain** or **local name space**. The domain is the collection of objects to which the process has access. A domain for a user at a given time might include some programs, files, data segments, and I/O devices such as a printer and a terminal. An example of a domain is shown in figure below.

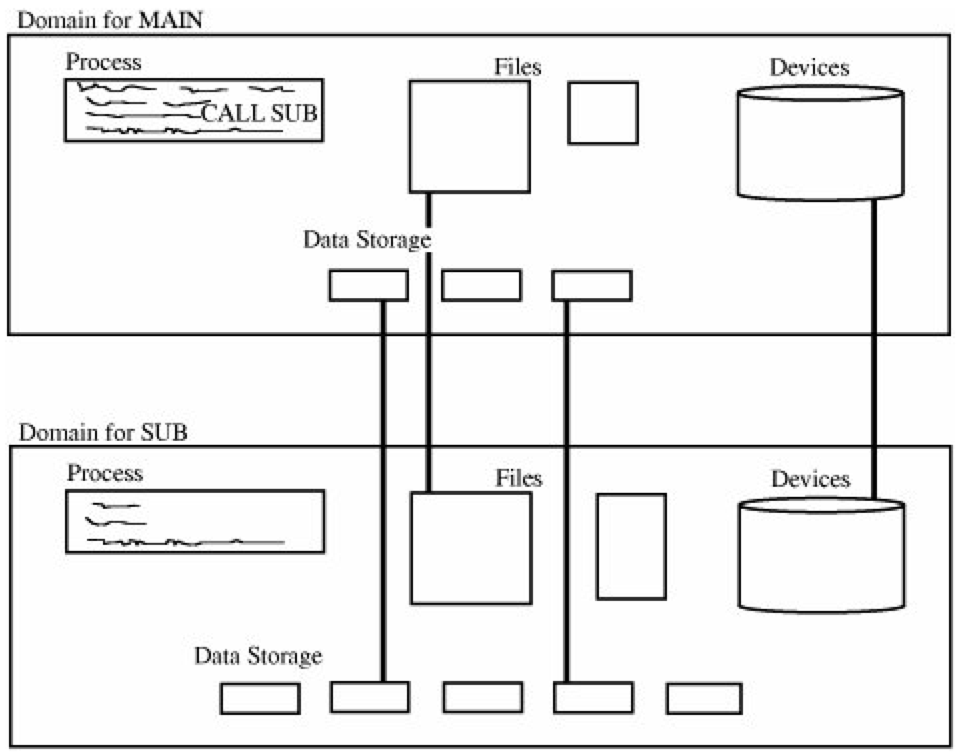
Figure 13. Process Execution Domain.



As execution continues, the process may call a subprocedure, passing some of the objects to which it has access as arguments to the subprocedure. The domain of the subprocedure is not necessarily the same as that of its calling procedure; in fact, a calling procedure may pass only some of its objects to the subprocedure, and the subprocedure may have access rights to other objects not accessible to the calling procedure. The caller may also pass only some of its access rights for the objects it passes to the subprocedure. For example, a procedure might pass to a subprocedure the right to read but not modify a particular data value.

Because each capability identifies a single object in a domain, the collection of capabilities defines the domain. When a process calls a subprocedure and passes certain objects to the subprocedure, the operating system forms a stack of all the capabilities of the current procedure. The operating system then creates new capabilities for the subprocedure, as shown in figure below.

Figure . 14. Passing Objects to a Subject.



Capabilities are a straightforward way to keep track of the access rights of subjects to objects during execution. The capabilities are backed up by a more comprehensive table, such as an access control matrix or an access control list. Each time a process seeks to use a new object, the operating system examines the master list of objects and subjects to determine whether the object is accessible. If so, the operating system creates a capability for that object.

Capabilities must be stored in memory inaccessible to normal users. One way of accomplishing this is to store capabilities in segments not pointed at by the user's segment table or to enclose them in protected memory as from a pair of base/bounds registers. Another approach is to use a tagged architecture machine to identify capabilities as structures requiring protection.

During execution, only the capabilities of objects that have been accessed by the current process are kept readily available. This restriction improves the speed with which access to an object can be checked. This approach is essentially the one used in Multics OS.

Capabilities can be revoked. When an issuing subject revokes a capability, no further access under the revoked capability should be permitted. A capability table can contain pointers to the active capabilities spawned under it so that the operating system can trace what access rights should be deleted if a capability is revoked. A similar problem is deleting capabilities for users who are no longer active.

Kerberos

Kerberos implements both authentication and access authorization by means of capabilities, called tickets, secured with symmetric cryptography. Kerberos requires two systems, called the authentication server (AS) and the ticket-granting server (TGS), which are both part of the key distribution center (KDC). A user presents an authenticating credential (such as a password) to the authentication server and receives a ticket showing that the user has passed authentication. The ticket must be encrypted to prevent the user from modifying or forging one claiming to be a different user, and the ticket must contain some provision to prevent one user from acquiring another user's ticket to impersonate that user.

Procedure-Oriented Access Control

One goal of access control is restricting not just which subjects have access to an object, but also what they can do to that object. Read versus write access can be controlled rather readily by most operating systems, but more complex control is not so easy to achieve.

By **procedure-oriented** protection, we imply the existence of a procedure that controls access to objects. In essence, the procedure forms a capsule around the object, permitting only certain specified accesses.

Procedures can ensure that accesses to an object be made through a trusted interface. For example, neither users nor general operating system routines might be allowed direct access to the table of valid users. Instead, the only accesses allowed might be through three procedures: one to add a user, one to delete a user, and one to check whether a particular name corresponds to a valid user. These procedures especially add and delete, could use their own checks to make sure that calls to them are legitimate.

Procedure-oriented protection implements the principle of information hiding because the means of implementing an object are known only to the object's control procedure. Of course, this degree of protection carries a penalty of inefficiency. With procedure-oriented protection, there can be no simple, fast access, even if the object is frequently used.

Role-Based Access Control

Role-based access control lets us associate privileges with groups, such as all administrators can do this or candlestick makers are forbidden to do this. Administering security is easier if we can control access by job demands, not by person. Access control keeps up with a person who changes responsibilities, and the system administrator does not have to choose the appropriate access control settings for someone.

File Protection Mechanism

The basic forms of protection are:

- All None protection
- Group protection

- Individual protection
- Per Object and Per User protection

All None Protection

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

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

The disadvantages for all none protection is

- Lack of trust: -In case of large environment, all users can’t be trusted.
- Too coarse: - If a user identifies a group of trustworthy users how to access these users is difficult.
- Rise of sharing: - In Batch systems user intervention is less whereas but in shared system user interact a lot.
- Complexity: - Since users have greater involvement, operating system performance is degraded.
- File listings: -Various system utilities tells us as to what all files are present ,if privilege given to all ,all users interact with sensitive files either.

Group Protection

All authorized users are separated into groups. A group may consist of several members working on a common project, a department, a class, or a single user. The basis for group membership is needed to share. The group members have some common interest and therefore are assumed to have files to share with the other group members. In this approach, no user belongs to more than one group.

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

The advantages of the group protection approach are:

- Easy to implement.
- A user is recognized by two identifiers: a user ID and a group ID.
    - These identifiers are stored in the file directory entry for each file and are obtained by the operating system when a user logs in.
    - Therefore, the operating system can easily check whether a proposed access to a file is requested from someone whose group ID matches the group ID for the file to be accessed.

The disadvantages of group protection mechanism are:

- Group affiliation: - A single user cannot belong to two groups. These ambiguities are most simply resolved by declaring that every user belongs to exactly one group. This restriction does not mean that all users belong to the same group.
- Multiple personalities: - To overcome the one-person one-group restriction, certain people might obtain multiple accounts, permitting them, in effect, to be multiple users. This hole in the protection approach leads to new problems because a single person can be only one user at a time. Multiple personalities lead to a proliferation of accounts,

redundant files, limited protection for files of general interest, and inconvenience to users.

- All groups: - To avoid multiple personalities, the system administrator may decide that Tom should have access to all his files any time he is active. This solution puts the responsibility on Tom to control with whom he shares what things.
- Limited sharing: - Files can be shared only within groups or with the world. Users want to be able to identify sharing partners for a file on a per-file basis; for example, sharing one file with ten people and another file with twenty others.

Individual Permissions

This can be of two types:

- **Persistent methods: -** Access acquired using a password, a token or ticket or by being in a access control list. User access permissions can be required for any access or only for modifications (write access). In this revocation of a privilege is challenging.

- **Temporary acquired permission: -** The UNIX designers added a permission called set userid (suid). If this protection is set for a file to be executed, the protection level is that of the file's owner, not the executor.

Per object per user protection

The primary limitation of these protection schemes is the ability to create meaningful groups of related users who should have similar access to related objects. The access control lists or access control matrices described earlier provide very flexible protection. Their disadvantage is for the user who wants to allow access to many users and to many different data sets; such a user must still specify each data set to be accessed by each user. As a new user is added, that user's special access rig hts must be specified by all appropriate users.

Models of Security

Security models are used to:

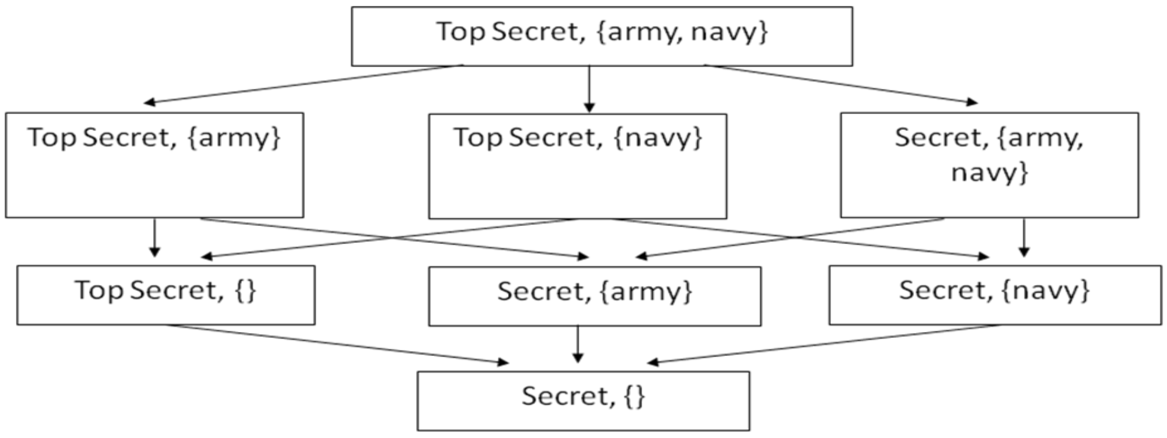
- test a particular policy for completeness and consistency
- document a policy
- help conceptualize and design an implementation
- check whether an implementation meets its requirements

Multilevel Security: - The capability of a computer system to carry information with different sensitivities (i.e. classified information at different security levels), permit simultaneous access by users with different security clearances and needs-to-know, and prevent users from obtaining access to information for which they lack authorization. Typically use Mandatory Access Control. Primary Security Goal is Confidentiality.

A lattice is a mathematical structure of elements organized by a relation among them, represented by a relational operator. A relation $\leq$ is called a partial ordering

- transitive: If $a \leq b$ and $b \leq c$, then $a \leq c$
- antisymmetric: If $a \leq b$ and $b \leq a$, then $a = b$

In a lattice, there may be elements a and b for which neither $a \leq b$ nor $b \leq a$. However, every pair of elements possesses an upper bound u and lowerbound l. Below shows an example for security lattice with levels={top secret, secret} and categories={army,navy}.



The Bell-La Padula Model

The Bell-La Padula model is a formal description of the allowable paths of information flow in a secure system. The model's goal is to identify allowable communication when maintaining secrecy is important. The model has been used to define security requirements for systems concurrently handling data at different sensitivity levels.

One purpose for security-level analysis is to enable us to construct systems that can perform concurrent computation on data at two different sensitivity levels. The programs processing top-secret data would be prevented from leaking top-secret data to the confidential data, and the confidential users would be prevented from accessing the top-secret data. Thus, the Bell-La Padula model is useful as the basis for the design of systems that handle data of multiple sensitivities.

Example of security levels

- Top Secret
- Secret
- Confidential
- Unclassified

Consider a security system with the following properties. The system covers a set of subjects S and a set of objects O. Each subject s in S and each object o in O has a fixed security class C(s) and C(o) (denoting clearance and classification level). The security classes are ordered by a relation ‘≤’ . Two properties characterize the secure flow of information.

Simple Security Property: - A subject s may have read access to an object o only if $C(o) \leq C(s)$.

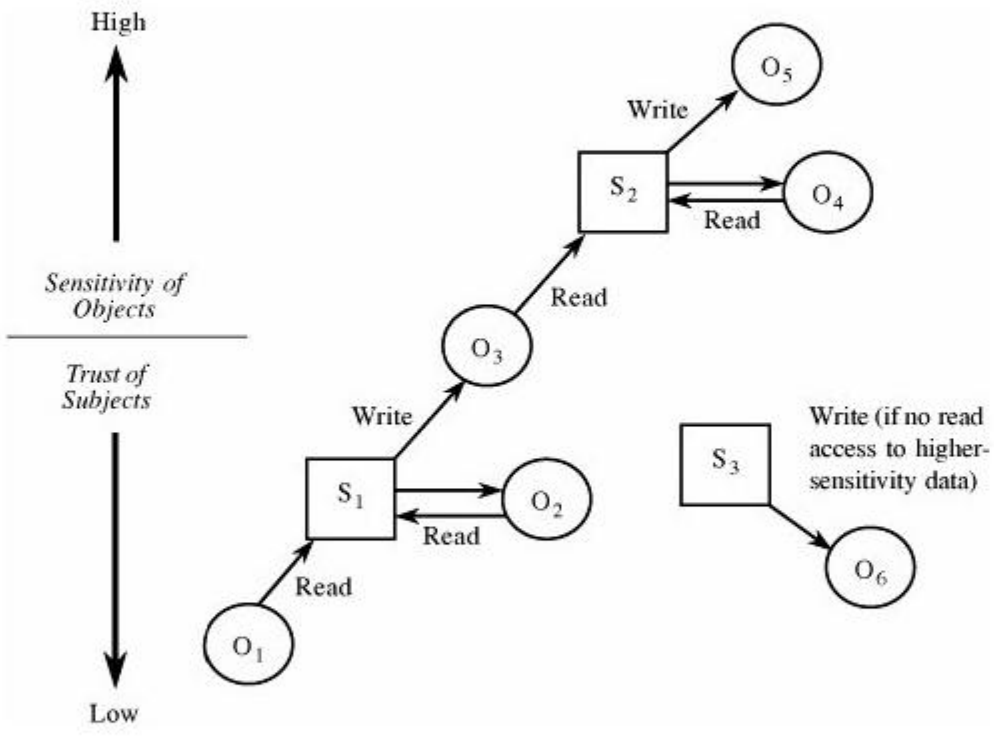
In the military model, this property says that the security class (clearance) of someone receiving a piece of information must be at least as high as the class (classification) of the information.

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

In the military model, one interpretation of the *-property is that a person obtaining information at one level may pass that information along only to people at levels no lower than the level of the information. The *-property prevents **write-down**, which occurs when a subject with access to high-level data transfers that data by writing it to a low-level object.

The implications of these two properties are shown in figure below. The classifications of subjects (represented by squares) and objects (represented by circles) are indicated by their positions: As the classification of an item increases, it is shown higher in the figure. The flow of information is generally horizontal (to and from the same level) and upward (from lower levels to higher). A downward flow is acceptable only if the highly cleared subject does not pass any high-sensitivity data to the lower-sensitivity object.

Figure 5-7. Secure Flow of Information.



The Biba Integrity Model

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

The Biba model is the counterpart (sometimes called the dual) of the BellLa Padula model. Biba defines "integrity levels," which are analogous to the sensitivity levels of the BellLa Padula model. Subjects and objects are ordered by an integrity classification scheme, denoted I(s) and I(o). The properties are

Simple Integrity Property: - Subject s can modify (have write access to) object o only if $I(s) \geq I(o)$

Integrity *-Property: - If subject s has read access to object o with integrity level I(o), s can have write access to object p only if $I(o) \geq I(p)$

This model addresses the integrity issue that the BellLa Padula model ignores. However, in doing so, the Biba model ignores secrecy. Secrecy-based security systems have been much more fully studied than have integrity-based systems. The current trend is to join secrecy and integrity concerns in security systems, although no widely accepted formal models achieve this compromise.

System Security

Intruders

One of the two most publicized threats to security is the intruder or hacker or cracker. Anderson identified three classes of intruders:

- Masquerader: An individual who is not authorized to use the computer and who penetrates a system's access controls to exploit a legitimate user's account
- Misfeasor: A legitimate user who accesses data, programs, or resources for which such access is not authorized, or who is authorized for such access but misuses his or her privileges

- Clandestine user: An individual who seizes supervisory control of the system and uses this control to evade auditing and access controls or to suppress audit collection

Masquerader is likely to be an outsider. Misfeasor generally is an insider and clandestine user can be either an outsider or an insider. The objective of the intruder is to gain access to a system or to increase the range of privileges accessible on a system. A system must maintain a file that associates a password with each authorized user. If such a file is stored with no protection, then it is an easy matter to gain access to it and learn passwords. The password file can be protected in one of two ways:

- One-way function: The system stores only the value of a function based on the user's password
- Access control: Access to the password file is limited to one or a very few accounts.

If one or both of these countermeasures are in place, some effort is needed for a potential intruder to learn passwords. On the basis of a survey of the literature and interviews with a number of password crackers, reports the following techniques for learning passwords

1. Try default passwords used with standard accounts that are shipped with the system. Many administrators do not bother to change these defaults.
2. Exhaustively try all short passwords (those of one to three characters).
3. Try words in the system’s online dictionary or a list of likely passwords. Examples of the latter are readily available on hacker bulletin boards.
4. Collect information about users, such as their full names, the names of their spouse and children, pictures in their office, and books in their office that are related to hobbies.
5. Try users’ phone numbers, Social Security numbers, and room numbers.
6. Try all legitimate license plate numbers for this state.
7. Use a Trojan horse to bypass restrictions on access.
8. Tap the line between a remote user and the host system.

Intrusion Detection

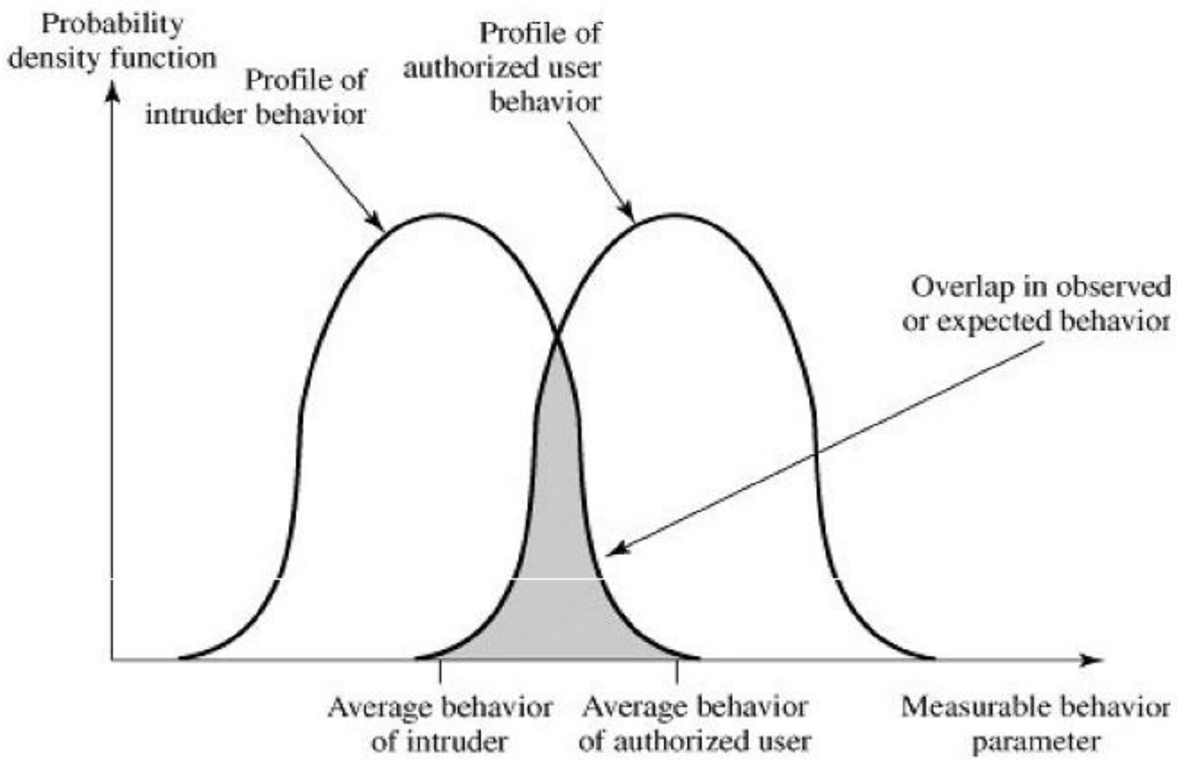
If an intrusion is detected quickly enough, the intruder can be identified and ejected from the system before any damage is done or any data are compromised. An effective intrusion detection system can serve as a deterrent, so acting to prevent intrusions. Intrusion detection enables the collection of information about intrusion techniques that can be used to strengthen the intrusion prevention facility. Intrusion detection is based on the assumption that the behavior of the intruder differs from that of a legitimate user in ways that can be quantified.

The following are approaches to intrusion detection:

1. Statistical anomaly detection: - Involves the collection of data relating to the behavior of legitimate users over a period of time. Then statistical tests are applied to observed behavior to determine with a high level of confidence whether that behavior is not legitimate user behavior.
   - Threshold detection: This approach involves defining thresholds, independent of user, for the frequency of occurrence of various events..
   - Profile based: A profile of the activity of each user is developed and used to detect changes in the behavior of individual accounts.
2. Rule-based detection: Involves an attempt to define a set of rules that can be used to decide that a given behavior is that of an intruder.
   - Anomaly detection: Rules are developed to detect deviation from previous usage patterns.
   - Penetration identification: An expert system approach that searches for suspicious behavior.

Statistical approaches attempt to define normal, or expected, behavior, whereas rule-based approaches attempt to define proper behavior.

Figure . Profiles of Behavior of Intruders and Authorized Users
(This item is displayed on page 571 in the print version)



Audit Records: - A fundamental tool for intrusion detection is the audit record . Some record of ongoing activity by users must be maintained as input to an intrusion detection system. Basically, two plans are used:

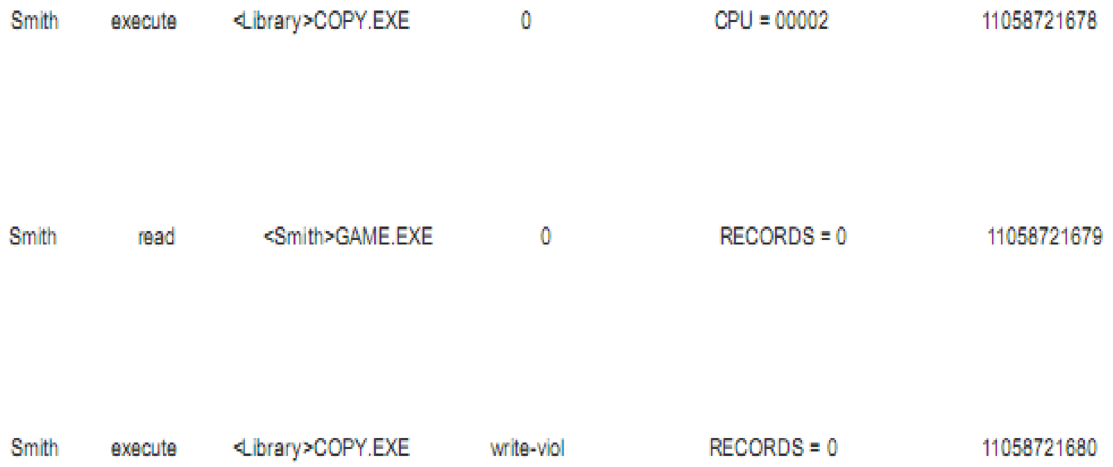
- Native audit records: Virtually all multiuser operating systems include accounting software that collects information on user activity.
- Detection-specific audit records: A collection facility can be implemented that generates audit records containing only that information required by the intrusion detection system.

A detection-specific audit records developed by Dorothy Denning contains the following fields:

- Subject: Initiators of actions.
- Action: Operation performed by the subject on or with an object; for example, login, read, perform I/O, execute.
- Object: Receptors of actions. Examples include files, programs, messages, records, terminals, printers, and user- or program-created structures.
- Exception-Condition: Denotes which, if any, exception condition is raised on return.

- Resource-Usage: A list of quantitative elements in which each element gives the amount used of some resource (e.g., number of lines printed or displayed, number of records read or written, processor time, I/O units used, session elapsed time).
- Time-Stamp: Unique time-and-date stamp identifying when the action took place.

Example of audit record entry for the operation COPY GAME.EXE TO <Library>GAME.EXE



Smith	execute	<Library>COPY.EXE	0	CPU = 00002	11058721678
Smith	read	<Smith>GAME.EXE	0	RECORDS = 0	11058721679
Smith	execute	<Library>COPY.EXE	write-viol	RECORDS = 0	11058721680

In this case, the copy is aborted because Smith does not have write permission to <Library>. The decomposition of a user operation into elementary actions has three advantages:

1. Because objects are the protectable entities in a system, the use of elementary actions enables an audit of all behavior affecting an object.
2. Single-object, single-action audit records simplify the model and the implementation.
3. Because of the simple, uniform structure of the detection-specific audit records, it may be relatively easy to obtain this information or at least part of it by a straightforward mapping from existing native audit records to the detection-specific audit records.

Statistical Anomaly Detection

Two broad categories: threshold detection and profile-based systems. Threshold detection involves counting the number of occurrences of a specific event type over an interval of time. If the count surpasses what is considered a reasonable number that one might expect to occur, then intrusion is assumed. Threshold analysis is a crude and ineffective detector of even moderately sophisticated attacks. Both the threshold and the time interval must be determined. Profile-based anomaly detection focuses on characterizing the past behavior of individual users or related groups of users and then detecting significant deviations. A profile may consist of a set of parameters, so that deviation on just a single parameter may not be sufficient in itself to signal an alert.

The foundation of this approach is an analysis of audit records. The audit records provide input to the intrusion detection function in two ways.

1. First, the designer must decide on a number of quantitative metrics that can be used to measure user behavior. An analysis of audit records over a period of time can be used to determine the activity profile of the average user. Thus, the audit records serve to define typical behavior.
2. Second, current audit records are the input used to detect intrusion. That is, the intrusion detection model analyzes incoming audit records to determine deviation from average behavior.

Examples of metrics those are useful for profile-based intrusion detection:

- Counter: A nonnegative integer that may be incremented but not decremented until it is reset by management action. Typically, a count of certain event types is kept over a particular period of time. Examples include the number of logins by a single user during an hour, the number of times a given command is executed during a single user session, and the number of password failures during a minute.
- Gauge: A nonnegative integer that may be incremented or decremented. Typically, a gauge is used to measure the current value of some entity. Examples include the number of logical connections assigned to a user application and the number of outgoing messages queued for a user process.
- Interval timer: The length of time between two related events. An example is the length of time between successive logins to an account.
- Resource utilization: Quantity of resources consumed during a specified period. Examples include the number of pages printed during a user session and total time consumed by a program execution.

On these general metrics, various tests can be performed to determine whether current activity fits within acceptable limits.

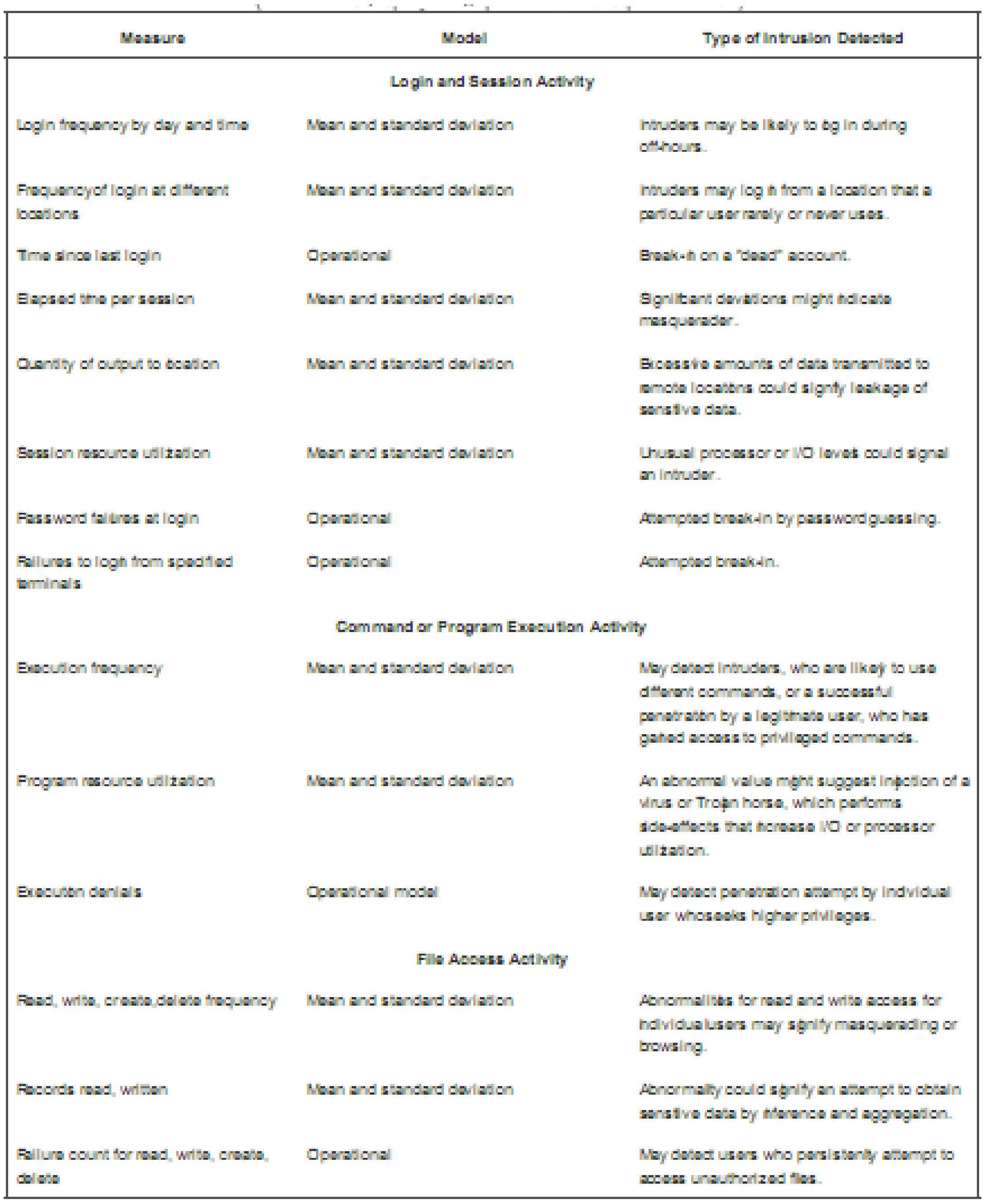
1. Mean and standard deviation
2. Multivariate
3. Markov process
4. Time series
5. Operational

Advantage of the use of statistical profiles:-

- A prior knowledge of security flaws is not required.
- The detector program learns what “normal” behavior is and then looks for deviations.
- The approach is not based on system-dependent characteristics and vulnerabilities.
- So it should be readily portable among a variety of systems.

Measures that can be used for the intrusion detection is shown in the below table.

Table . Measures That May Be Used for Intrusion Detection



Measure	Model	Type of Intrusion Detected
	Login and Session Activity	
Login frequency by day and time	Mean and standard deviation	Intruders may be likely to log in during off-hours.
Frequencyof login at different locations	Mean and standard deviation	Intruders may log in from a location that a particular user rarely or never uses.
Time since last login	Operational	Break-in on a "dead" account.
Elapsed time per session	Mean and standard deviation	Significant deviations might indicate masquerader.
Quantity of output to location	Mean and standard deviation	Excessive amounts of data transmitted to remote locations could signify leakage of sensitive data.
Session resource utilization	Mean and standard deviation	Unusual processor or I/O levels could signal an intruder.
Password failures at login	Operational	Attempted break-in by password guessing.
Failures to login from specified terminals	Operational	Attempted break-in.
	Command or Program Execution Activity	
Execution frequency	Mean and standard deviation	May detect intruders, who are likely to use different commands, or a successful penetration by a legitimate user, who has gained access to privileged commands.
Program resource utilization	Mean and standard deviation	An abnormal value might suggest injection of a virus or Trojan horse, which performs side-effects that increase I/O or processor utilization.
Execution denials	Operational model	May detect penetration attempt by individual user who seeks higher privileges.
	File Access Activity	
Read, write, create, delete frequency	Mean and standard deviation	Abnormalities for read and write access for individual users may signify masquerading or browsing.
Records read, written	Mean and standard deviation	Abnormality could signify an attempt to obtain sensitive data by inference and aggregation.
Failure count for read, write, create, delete	Operational	May detect users who persistently attempt to access unauthorized files.

Rule-Based Intrusion Detection

Rule-based techniques detect intrusion by observing events in the system and applying a set of rules that lead to a decision regarding whether a given pattern of activity is or is not suspicious. We can characterize all approaches as focusing on either anomaly detection or penetration identification, although there is some overlap in these approaches. Rule-based anomaly detection is similar in terms of its approach and strengths to statistical anomaly detection. With the rule-based approach, historical audit records are analyzed to identify usage patterns and to generate automatically rules that describe those patterns. Rules may represent past behavior patterns of users, programs, privileges, time slots, terminals, and so on. Current behavior is then observed, and each transaction is matched against the set of rules to determine if it conforms to any historically observed pattern of behavior.

As with statistical anomaly detection, rule-based anomaly detection does not require knowledge of security vulnerabilities within the system. Rather, the scheme is based on observing past behavior and, in effect, assuming that the future will be like the past. In order for this approach to be effective, a rather large database of rules will be needed.

Rule-based penetration identification takes a very different approach to intrusion detection, one based on expert system technology. The key feature of such systems is the use of rules for identifying known penetrations or penetrations that would exploit known weaknesses. Rules can also be defined that identify suspicious behavior, even when the behavior is within the bounds of established patterns of usage. Typically, the rules used in these systems are specific to the machine and operating system. Also, such rules are generated by "experts" rather than by means of an automated analysis of audit records. The normal procedure is to interview system administrators and security analysts to collect a suite of known penetration scenarios and key events that threaten the security of the target system. Thus, the strength of the approach depends on the skill of those involved in setting up the rules.

A simple example of the type of rules that can be used is found in NIDX

1. Users should not read files in other users' personal directories.
2. Users must not write other users' files.
3. Users who log in after hours often access the same files they used earlier.
4. Users do not generally open disk devices directly but rely on higher-level operating system utilities.
5. Users should not be logged in more than once to the same system.
6. Users do not make copies of system programs.

The Base-Rate Fallacy:- An intrusion detection system should detect a substantial percentage of intrusions while keeping the false alarm rate at an acceptable level. If only a modest percentage of actual intrusions are detected, the system provides a false sense of security. If the system frequently triggers an alert when there is no intrusion (a false alarm), then either system managers will begin to ignore the alarms, or much time will be wasted analyzing the false alarms.

Distributed Intrusion Detection

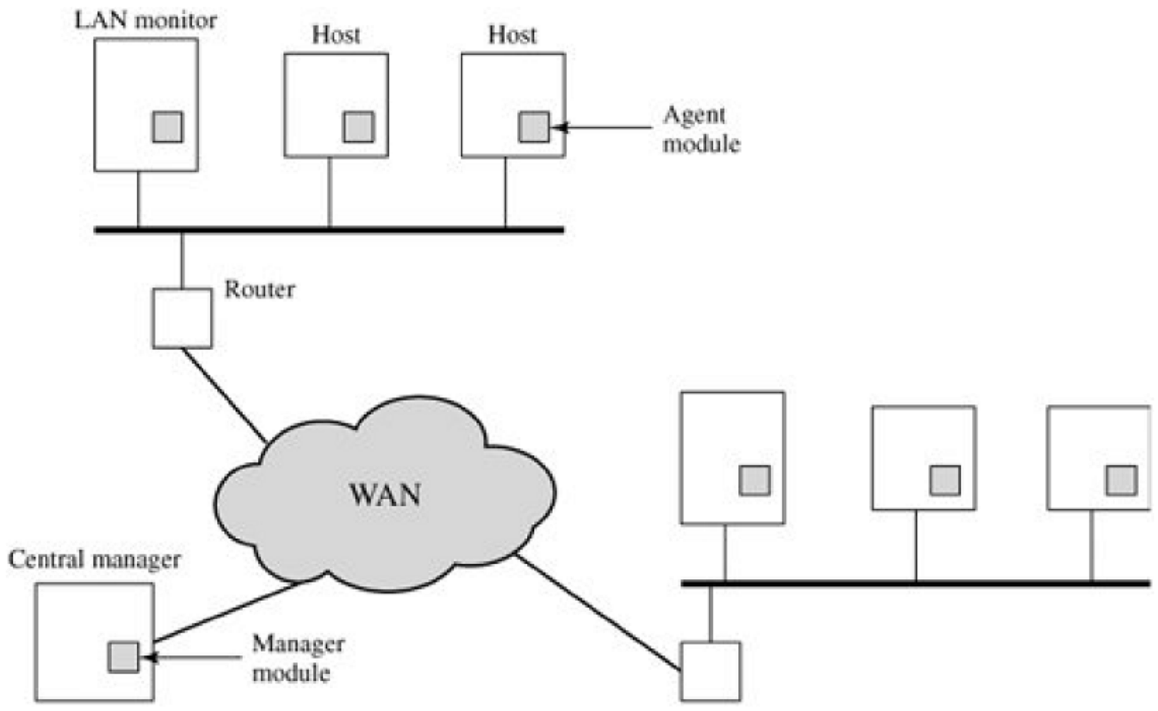
Intrusion detection systems focused on single-system stand-alone facilities. The typical organizations have a distributed collection of hosts supported by a LAN or internetwork. Although it is possible to mount a defense by using stand-alone intrusion detection systems on each host, a more effective defense can be achieved by coordination and cooperation among intrusion detection systems across the network.

Issues in the design of distributed IDS

- A distributed intrusion detection system may need to deal with different audit record formats.
- One or more nodes in the network will serve as collection and analysis points for the data from the systems on the network. Thus, either raw audit data or summary data must be transmitted across the network.
- Either a centralized or decentralized architecture can be used.

Below figure shows architecture for Distributed Intrusion Detection. The main components are:-

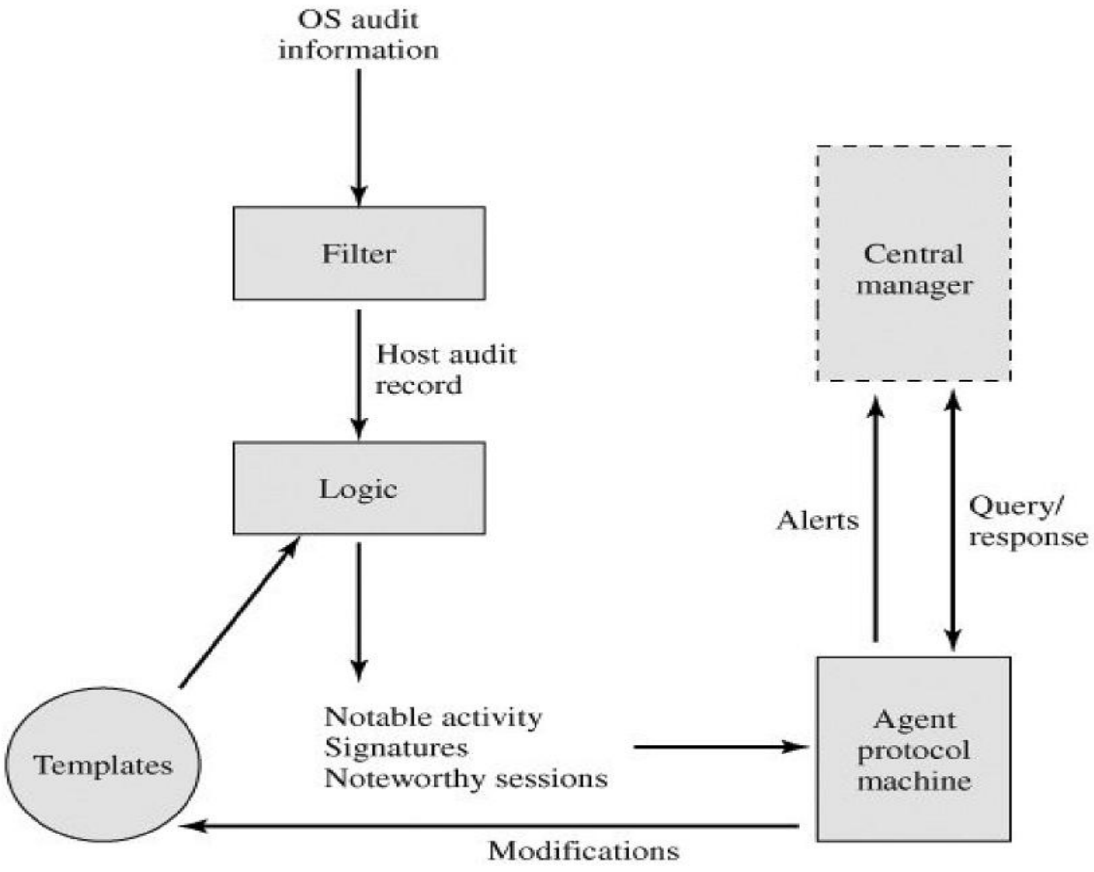
- Host agent module: An audit collection module operating as a background process on a monitored system. Its purpose is to collect data on security-related events on the host and transmit these to the central manager.
- LAN monitor agent module: Operates in the same fashion as a host agent module except that it analyzes LAN traffic and reports the results to the central manager.
- Central manager module: Receives reports from LAN monitor and host agents and processes and correlates these reports to detect intrusion.



The scheme is designed to be independent of any operating system or system auditing implementation. The agent captures each audit record produced by the native audit collection system. A filter is applied that retains only those records that are of security interest. These records are then reformatted into a standardized format referred to as the host audit record (HAR). A template-driven logic module analyzes the records for suspicious activity.At the lowest level, the agent scans for notable events that are of interest independent of any past events. Examples include failed file accesses, accessing system files,and changing a file's access control. At the next higher level, the agent looks for sequences of events, such as known attack patterns (signatures). Finally, the agent looks for anomalous behavior of an individual user based on a historical profile of that user, such as number of programs executed, number of files accessed, and the like.

The Agent architecture is shown below. When suspicious activity is detected, an alert is sent to the central manager. The central manager includes an expert system that can draw inferences from received data. The manager may also query individual systems for copies of HARs to correlate with those from other agents. The LAN monitor agent also supplies information to the central manager. The LAN monitor agent audits host-host connections, services used, and volume of traffic. It searches for significant events, such as sudden changes in network load, the use of security-related services, and network activities such as rlogin.

Figure . Agent Architecture



Honeypots:- Honeypots are decoy systems that are designed to lure a potential attacker away from critical systems. Honeypots are designed to

1. divert an attacker from accessing critical systems
2. collect information about the attacker's activity
3. encourage the attacker to stay on the system long enough for administrators to respond

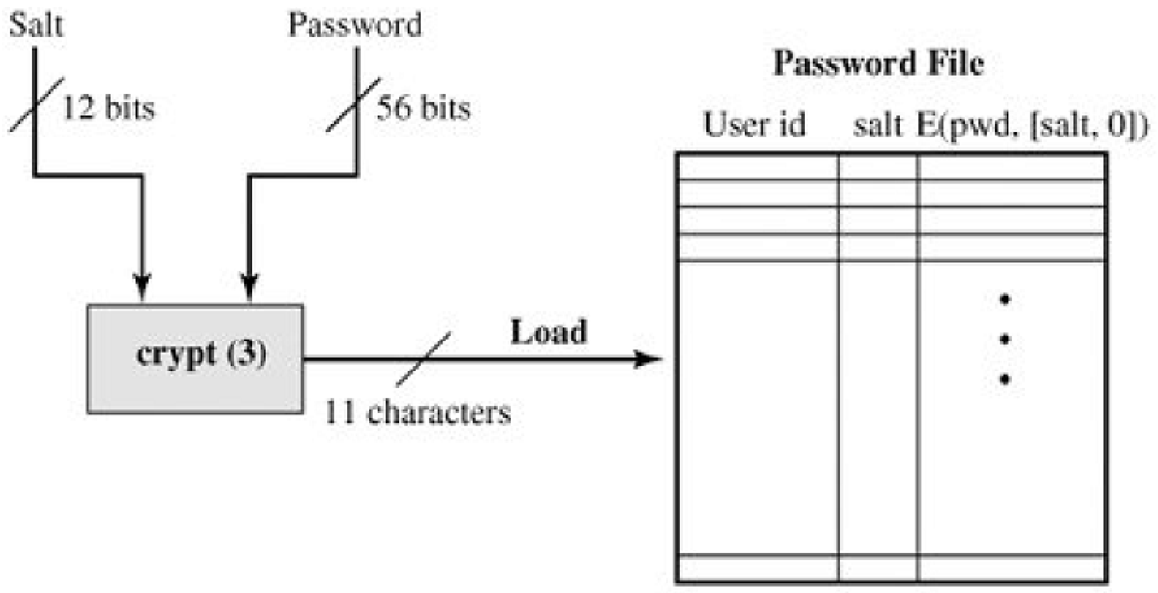
Intrusion Detection Exchange Format:- A requirements document, which describes the high-level functional requirements for communication between intrusion detection systems and requirements for communication between intrusion detection systems and with management systems, including the rationale for those requirements. Scenarios will be used to illustrate the requirements. A common intrusion language specification, which describes data formats that satisfy the requirements. A framework document, which identifies existing protocols best used for communication between intrusion detection systems, and describes how the devised data formats relate to them.

Password Management

ID provides security in the following ways:

- The ID determines whether the user is authorized to gain access to a system. In some systems, only those who already have an ID filed on the system are allowed to gain access.
- The ID determines the privileges accorded to the user. A few users may have supervisory or "superuser" status that enables them to read files and perform functions that are especially protected by the operating system. Some systems have guest or anonymous accounts, and users of these accounts have more limited privileges than others.
- The ID is used in what is referred to as discretionary access control. For example, by listing the IDs of the other users, a user may grant permission to them to read files owned by that user.

Consider a scheme that is widely used on UNIX, in which passwords are never stored in the clear which is shown below. Each user selects a password of up to eight printable characters in length. This is converted into a 56-bit value (using 7-bit ASCII) that serves as the key input to an encryption routine. The encryption routine, known as crypt(3), is based on DES. The DES algorithm is modified using a 12-bit " salt " value. Typically, this value is related to the time at which the password is assigned to the user. The modified DES algorithm is exercised with a data input consisting of a 64-bit block of zeros. The output of the algorithm then serves as input for a second encryption. This process is repeated for a total of 25 encryptions. The resulting 64-bit output is then translated into an 11-character sequence. The hashed password is then stored, together with a plaintext copy of the salt, in the password file for the corresponding user ID. This method has been shown to be secure against a variety of cryptanalytic attacks.

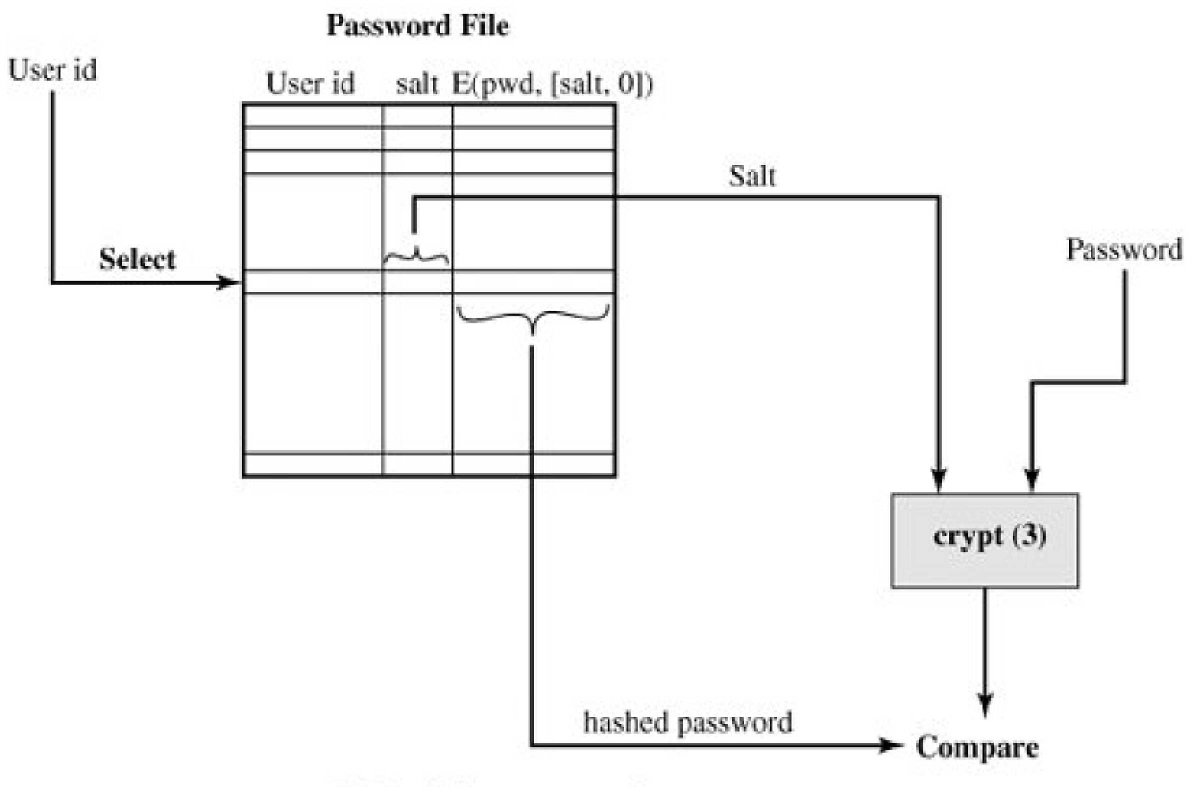


(a) Loading a new password

The salt serves three purposes:

- It prevents duplicate passwords from being visible in the password file. Even if two users choose the same password, those passwords will be assigned at different times. Hence, the "extended" passwords of the two users will differ.
- It effectively increases the length of the password without requiring the user to remember two additional characters. Hence, the number of possible passwords is increased by a factor of 4096, increasing the difficulty of guessing a password.
- It prevents the use of a hardware implementation of DES, which would ease the difficulty of a brute-force guessing attack.

When a user attempts to log on to a UNIX system, the user provides an ID and a password. The operating system uses the ID to index into the password file and retrieve the plaintext salt and the encrypted password. The salt and user-supplied passwords are used as input to the encryption routine. If the result matches the stored value, the password is accepted.



(b) Verifying a password

There are two threats to the UNIX password scheme.

- First, a user can gain access on a machine using a guest account or by some other means and then run a password guessing program, called a password cracker, on that machine. The attacker should be able to check hundreds and perhaps thousands of possible passwords with little resource consumption.
- In addition, if an opponent is able to obtain a copy of the password file, then a cracker program can be run on another machine at leisure. This enables the opponent to run through many thousands of possible passwords in a reasonable period.

Password crackers rely on the fact that some people choose easily guessable passwords. A simple remedy is for the system to reject any password choice of fewer than, say, six characters or even to require that all passwords be exactly eight characters in length. Most users would not complain about such a restriction.

- Following strategy were used for guessing One-fourth of the passwords. Try the user's name, initials, account name, and other relevant personal information. In all, 130 different permutations for each user were tried.

- Try words from various dictionaries. The author compiled a dictionary of over 60,000 words, including the online dictionary on the system itself, and various other lists as shown.
- Try various permutations on the words from step 2. This included making the first letter uppercase or a control character, making the entire word uppercase, reversing the word, changing the letter "o" to the digit "zero," and so on. These permutations added another 1 million words to the list.
- Try various capitalization permutations on the words from step 2 that were not considered in step 3. This added almost 2 million additional words to the list.

Access Control: - One way to thwart a password attack is to deny the opponent access to the password file. If the encrypted password portion of the file is accessible only by a privileged user, then the opponent cannot read it without already knowing the password of a privileged user.

Password Selection Strategies: - Our goal, then, is to eliminate guessable passwords while allowing the user to select a password that is memorable. Four basic techniques are in use:

- User education
- Computer-generated passwords
- Reactive password checking
- Proactive password checking

The possible approaches to proactive password checking are a simple system for rule enforcement. For example, the following rules could be enforced:

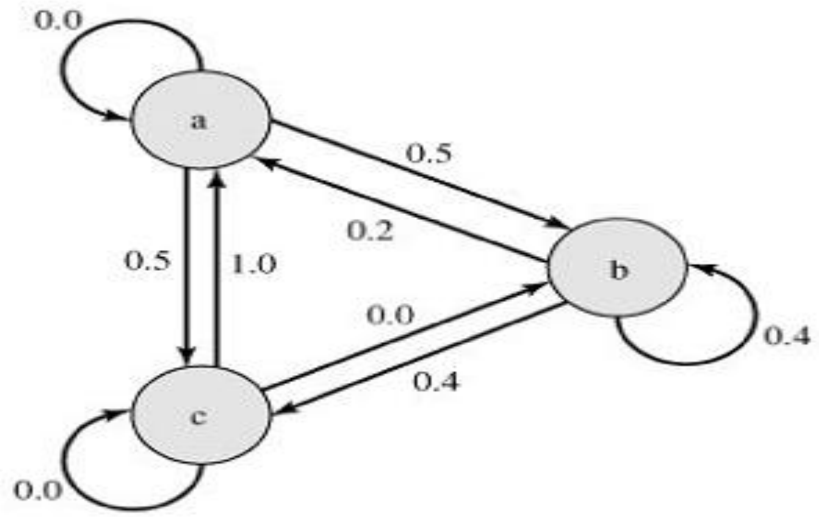
- ➢ All passwords must be at least eight characters long.
- ➢ In the first eight characters, the passwords must include at least one each of uppercase, lowercase, numeric digits, and punctuation marks.

Another possible procedure is simply to compile a large dictionary of possible "bad" passwords. When a user selects a password, the system checks to make sure that it is not on the disapproved list. There are two problems with this approach:

- ➢ Space: The dictionary must be very large to be effective
- ➢ Time: The time required to search a large dictionary may itself be large.

Two techniques for developing an effective and efficient proactive password checker that is based on rejecting words on a list show promise. One of these develops a Markov model for the generation of guessable passwords. The figure below shows a simplified version of such a model. This model shows a language consisting of an alphabet of three characters. The state of the system at any time is the identity of the most recent letter. The value on the transition from one state to another represents the probability that one letter follows another. Thus, the probability that the next letter is b, given that the current letter is a, is 0.5.

Figure 18.5. An Example Markov Model



$M = \{3, \{a, b, c\}, T, 1\}$ where

$$T = \begin{bmatrix} 0.0 & 0.5 & 0.5 \\ 0.2 & 0.4 & 0.4 \\ 1.0 & 0.0 & 0.0 \end{bmatrix}$$

e.g., string probably from this language: abbcacaba

e.g., string probably not from this language: aacccbaaa

Viruses and Related Threats

Malicious Programs

Malicious programs threats can be divided into two categories:

- Those that need a host program,
- Those that is independent.

The former are essentially fragments of programs that cannot exist independently of some actual application program, utility, or system program. The latter are self-contained programs that can be scheduled and run by the operating system.

We can also differentiate as

- Software threats that do not replicate
- Software threats that replicates

The former are fragments of programs that are to be activated when the host program is invoked to perform a specific function. The latter consist of either a program fragment (virus) or an independent program (worm, bacterium) that, when executed, may produce one or more copies of itself to be activated later on the same system or some other system.

Trap doors

A trap door is a secret entry point into a program that allows someone that is aware of the trap door to gain access without going through the procedures. Trap doors have been used legitimately for many years by programmers to debug and test programs. This usually is done when the programmer is developing an application that has an authentication procedure, or a long setup, requiring the user to enter many different values, to run the application. To debug the program, the developer may wish to gain special privileges or to avoid all the necessary setup and authentication. The programmer may also want to ensure that there is a method of activating the program should something be wrong with the authentication procedure that is being built into the application. The trap door is code that recognizes some special sequence of input or is triggered by being run from a certain user ID or by an unlikely sequence of events.

Trap doors become threats when they are used by unscrupulous programmers to gain unauthorized access. It is difficult to implement operating system controls for trap doors. Security measures must focus on the program development and software update activities.

Logic Bomb

One of the oldest types of program threat, predating viruses and worms, is the logic bomb. The logic bomb is code embedded in some legitimate program that is set to “explode” when certain conditions are met. Examples of conditions that can be used as triggers for a logic bomb are the presence or absence of certain files, a particular day of the week or date, or a particular user running the application.eg: a logic bomb checked for a certain employee ID numb and then triggered if the ID failed to appear in two consecutive payroll calculations. Once triggered, a bomb may alter or delete data or entire files, cause a machine halt, or do some other damage.

Trojan Horses

A Trojan horse is a useful program or command procedure containing hidden code that, when invoked, performs some unwanted or harmful function.
Trojan horse programs can be used to accomplish functions indirectly that an unauthorized user could not accomplish directly. For example, to gain access to the files of another user on a shared system, a user could create a Trojan horse program that, when executed, changed the invoking user's file permissions so that the file' are readable by any user. The author could then induce users to run the program by placing it in a common directory and naming it such that it appears to be a useful utility. An example is a program that ostensibly produces a listing of the user's files in a desirable format. After another user has run the program, the author can then access the information in the user's files.

Another common motivation for the Trojan horse is data destruction. The program appears to be performing a useful function (e.g., a calculator program), but it may also be quietly deleting the user's files or bulletin board system.

Viruses

A virus is a program that can "infect" other programs by modifying them: the modification includes a copy of the virus program, which can then go on to infect other programs.

A computer virus carries in its instructional code the recipe for making perfect copies of itself. Lodged in a host computer, the typical virus takes temporary control of the computer's disk operating system. Then, whenever the infected computer comes into contact with an uninfected piece of software, a fresh copy of the virus passes into the new program. Thus, the infection can be spread from computer to computer by unsuspecting users, who either swap disks or send programs to one another over a network. In a network environment, the ability to access applications and system services on other computers provides a perfect culture for the spread of a virus.

The Nature of Viruses

A virus can do anything that other programs do. The only difference is that it attaches itself to another program and executes secretly when the host program is run. Once a virus is executing, it can perform any function, such as erasing files and programs. During its lifetime, a typical virus goes through the following four stages

Dormant phase

The virus is idle. The virus will eventually be activated by some event, such as a date, the presence of another program or file, or the capacity of the disk exceeding some limit. Not all viruses have this stage.

Propagation phase

The virus places an identical copy of itself into other programs or into certain system areas on the disk. Each infected program will now contain a clone of the virus, which will itself enter a propagation phase.

Triggering phase

The virus is activated to perform the function for which it was intended. As with the dormant phase, the triggering phase can be caused by a variety of system events, including a count of the number of times that this copy of the virus has made copies of itself.

Execution phase

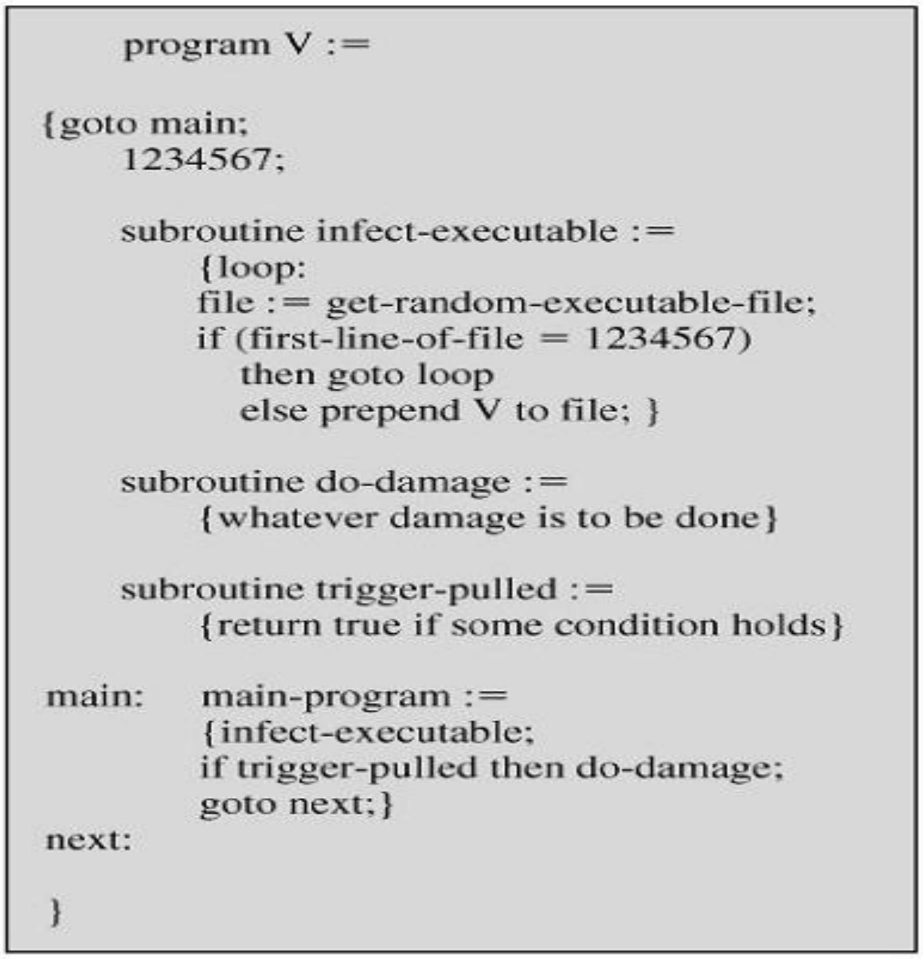
The function is performed. The function may be harmless, such as a message on the screen, or damaging, such as the destruction of programs and data files.
Most viruses carry out their work in a manner that is specific to a particular operating system

and, in some cases, specific to a particular hardware platform. Thus, they are designed to take advantage of the details and weaknesses of particular systems.

Virus Structure

A virus can be prepended or postpended to an executable program, or it can be embedded in some other fashion. The key to its operation is that the infected program, when invoked, will first execute the virus code and then execute the original code of the program.

Figure 19.1. A Simple Virus

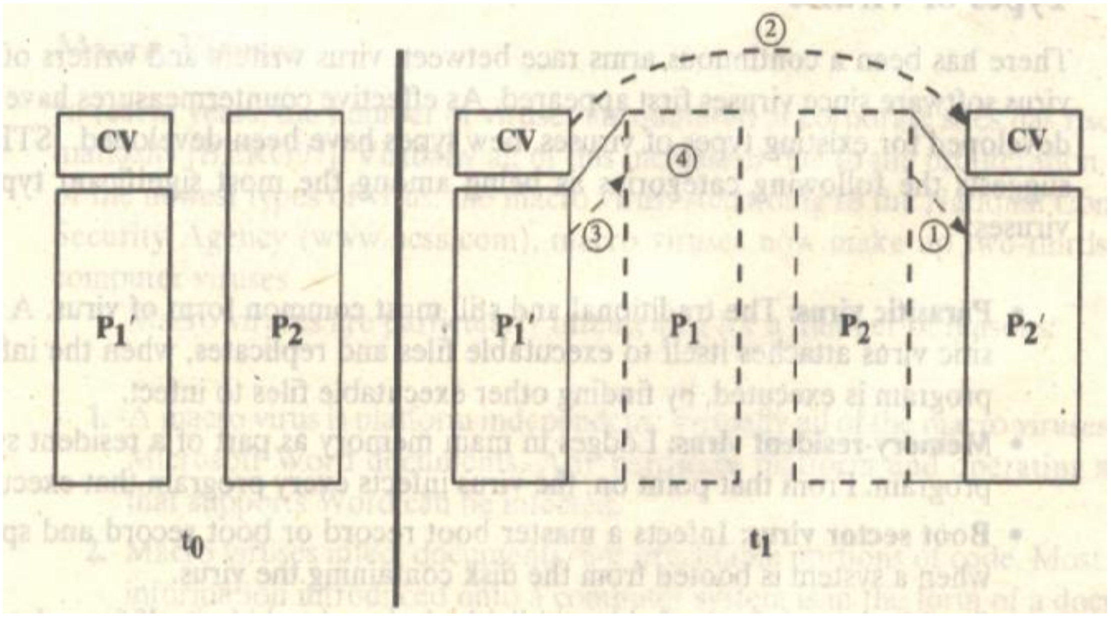


```
      program V :=

{goto main;
      1234567;

      subroutine infect-executable :=
            {loop:
            file := get-random-executable-file;
            if (first-line-of-file = 1234567)
               then goto loop
               else prepend V to file; }

      subroutine do-damage :=
            {whatever damage is to be done}

      subroutine trigger-pulled :=
            {return true if some condition holds}

main:       main-program :=
            {infect-executable;
            if trigger-pulled then do-damage;
            goto next;}
next:

}
```

A very general depiction of virus structure is shown in Figure . In this case, the virus code, V, is prepended to infected programs, and it is assumed that the entry point to the program, when invoked, is the first line of the program. An infected program begins with the virus code and works as follows. The first line of code is a jump to the main virus program. The second line is a special marker that is used by the virus to determine whether or not a potential victim program has already been infected with this virus. When the program is invoked, control is immediately transferred to the main virus program. The virus program first seeks out uninfected executable files and infects them. Next, the virus may perform some action, usually detrimental to the system. This action could be performed every time the program is invoked, or it could be a logic bomb that triggers only under certain conditions. Finally, the virus transfers control to the original program. If the infection phase of the program is reasonably rapid, a user is unlikely to notice any difference between the execution of an infected and uninfected program.

A virus such as the one just described is easily detected because an infected version of a program is longer than the corresponding uninfected one. A way to thwart such a simple means of detecting a virus is to compress the executable file so that both the infected and uninfected versions are of identical length. We assume that program P1 is infected with the virus CV. When this program is invoked, control passes to its virus, which performs the following steps:

- For each uninfected file P2 that is found, the virus first compresses that file to produce P’2, which is shorter than the original program by the size of the virus.
- A copy of the virus is prepended to the compressed program.
- The compressed version of the original infected program, P’1, is uncompressed.
- The uncompressed original program is executed.



Initial Infection

Once a virus has gained entry to a system by infecting a single program, it is in a position to infect some or all other executable files on that system when the infected program executes. Thus, viral infection can be completely prevented by preventing the virus from gaining entry in the first place. Most viral infections initiate with a disk from which programs are copied onto a machine. Many of these are disks that have games or simple but handy utilities that employees obtain for their home computers and then bring in and put on an office machine. Some are present on disks that come shrink-wrapped from the manufacturer of an application. Only a small fraction of infections begin across a network connection. Most of these are obtained from an electronic bulletin board system.

Types of Viruses

The main types of virus are,

Parasitic Virus

It is a virus that attaches itself to executable files and replicates, when the infected program is executed, by finding other executable files to infect.

Memory-resident virus

It lodges in main memory as part of a resident system program. From that point on, the virus infects every program that executes.

Boot sector virus

It infects a master boot record or boot record and spreads when a system is booted from the disk containing the virus.

Stealth virus

A form of virus explicitly designed to hide itself from detection by antivirus software.

Polymorphic virus

It is a virus that mutates with every infection, making detection by the "signature" of the virus impossible. One example of a stealth virus: a virus that uses compression so that the infected program is exactly the same length as an uninfected version.

A polymorphic virus creates copies during replication that are functionally equivalent but have distinctly different bit patterns. As with a stealth virus, the purpose is to defeat programs hat scan for viruses. In this case, the "signature" of the virus will vary with each copy. To achieve this variation, the virus may randomly insert superfluous instructions or interchange the order of independent instructions. A more effective approach is to use encryption. A portion of the virus, generally called a *mutation engine,* creates a random encryption key to encrypt the remainder of the virus. The key is stored with the virus, and the mutation engine itself is altered. When an infected program is invoked, the virus uses the stored random key to decrypt the virus. When the virus replicates, a different random key is selected.

Another weapon in the virus writers' armory is the virus-creation toolkit. Such a toolkit enables a relative novice to create quickly a number of different viruses. Although viruses created with toolkits tend to be less sophisticated than viruses designed from scratch, the sheer number of new viruses that can be generated creates a problem for antivirus schemes.

Yet another tool of the virus writer is the virus exchange bulletin board. A number of such boards have sprung up in the United States and other countries. These boards offer copies of viruses that can be downloaded, as well as tips for the creation of viruses.

Macro Viruses

According to the National Computer Security Agency (www.ncsa.com), macro viruses now make up two-thirds of all computer viruses. Macro viruses are particularly threatening for a number of reasons:

- A macro virus is platform independent. Virtually all of the macro viruses infect Microsoft Word documents. Any hardware platform and operating system that supports Word can be infected.
- Macro viruses infect documents, not executable portions of code. Most of the information introduced onto a computer system is in the form of a document rather than a program.
- Macro viruses are easily spread. A very common method is by electronic mail.

Macro viruses take advantage of a feature found in Word and other office applications such as Microsoft Excel, namely the macro. In essence, a macro is an executable program embedded in a word processing document or other type of file. Typically, users employ macros to automate repetitive tasks and thereby save keystrokes. The macro language is usually some form of the Basic programming language. A user might define a sequence of keystrokes in a macro and set it up so that the macro is invoked when a function key or special short combination of keys is input. What makes it possible to create a macro virus is the auto executing macro. This is a macro that is automatically invoked, without explicit user input. Common auto execute events are opening a file, closing a file, and starting an application. Once a macro is running, it can copy itself to other documents, delete files, and cause other sorts of damage to the user’s system. In Microsoft Word, there are three types of auto executing macros.

Auto execute

If a macro named AutoExec is in the "normal.dot" template or in a global template stored in Word's startup directory, it is executed whenever Word is started.

Auto macro

An auto macro executes when a defined event occurs, such as opening or closing a document, creating a new document, or quitting Word.

Command macro

If a macro in a global macro file or a macro attached to a document has the name of an existing Word command, it is executed whenever the user invokes that command

A common technique for spreading a macro virus is as follows. An auto macro or command macro is attached to a Word document that is introduced into a system by e-mail or disk transfer. At some point after the document is opened, the macro executes. The macro copies itself to the global macro file. When the next session of Word opens, the infected global macro is active. When this macro executes, it can replicate itself and cause damage. Successive releases of Word provide increased protection against macro viruses.

Worms

Network worm programs use network connections to spread from system to system. Once active within a system, a network worm can behave as a computer virus or bacteria, or it could implant Trojan horse programs or perform any number of disruptive or destructive actions.

To replicate itself, a network worm uses some sort of network vehicle. Examples include the following

- Electronic mail **facility:** A worm mails a copy of itself to other systems
- Remote execution capability A worm executes a copy of itself on another system.

• Remote login capability: A worm logs onto a remote system as a user and then uses commands to copy itself from one system to the other.

The new copy of the worm program is then run on the remote system where, in addition to any functions that it performs at that system, it continues to spread in the same fashion.

A network worm exhibits the same characteristics as a computer virus:

• Dormant phase

• Propagation phase

• Triggering phase

• Execution phase

The propagation phase generally performs the following functions:

• Search for other systems to infect by examining host tables or similar repositories of remote system addresses.

• Establish a connection with a remote system.

• Copy itself to the remote system and cause the copy to be run.

The network worm may also attempt to determine whether a system has previously been infected before copying itself to the system.

The Morris Worm

Until the current generation of worms, the best known was the worm released onto the Internet by Robert Morris in 1998. The Morris worm was designed to spread on UNIX systems and used a number of different techniques for propagation. When a copy began execution, its first task was to discover other hosts to this host that would allow entry from this host. The worm performed this task by examining a variety of lists and tables, including system tables that declared which other machines were trusted by this host, users' mail forwarding files, tables by which users gave themselves permission for access to remote accounts, and from a program that reported the status of network connections. For each discovered host, the worm tried a number of methods for gaining access:

1. It attempted to log on to a remote host as a legitimate user. In this method, the worm first attempted to crack the local password file, and then used the discovered passwords and corresponding user IDs. The assumption was that many users would use the same password on different systems. To obtain the passwords, the worm ran a password-cracking program that tried

a. Each user's account name and simple permutations of it

b. **A** list of 432 built-in passwords that Morris thought to be likely candidates

c. All the words in the local system directory

2. It exploited a bug in the finger protocol, which reports the whereabouts of a remote user.

3. It exploited a trapdoor in the debug option of the remote process that receives and sends mail.

4. If any of these attacks succeeded, the worm achieved communication with the operating system command interpreter. It then sent this interpreter a short bootstrap program, issued a command to execute that program, and then logged off. The bootstrap program then called back the parent program and downloaded the remainder of the worm. The new worm was then executed.

Recent Worm Attacks

The contemporary era of worm threats began with the release of the Code Red worm in July of 2001. Code Red exploits a security hole in the Microsoft Internet Information Server (IIS) to penetrate and spread. It also disables the system file checker in Windows. The worm probes random IP addresses to Spread to other hosts. During a certain period of time, it only spreads. It then initiates a denial-of-service attack against a government Web site by flooding the site with packets from numerous hosts. The worm then suspends activities and reactivates periodically. In the second wave of attack, Code Red infected nearly 360,000 servers in 14 hours. In addition to the havoc it causes at the targeted server, Code Red can consume enormous amounts of Internet capacity, disrupting service.

Code Red II is a variant that targets Microsoft IIS. In addition, this newer Worm installs a backdoor allowing a hacker to direct activities of victim computers.
In late 2001, a more versatile worm appeared, known as Nimda. Nimda spreads by multiple mechanisms

- From client to client via email
- From client to client via open network shares
- From web server to client via browsing of compromised website
- From client to web server via active scanning for and exploitation of various Microsoft IIS 4.0/5.0 directory traversal vulnerabilities
- From client to web server via scanning for the back doors left behind by the “Code Red II” worms

The worm modifies web documents and certain executable files found on the system it infects and create numerous copies of it under various filenames.

Virus Countermeasures

The main commonly used virus countermeasure is Antivirus Approaches. The ideal solution to the threat of viruses is prevention: Do not allow a virus to get into the system in the first place. The next best approach is to be able to do the following:

1. Detection
2. Identification
3. Removal

If detection succeeds but either identification or removal is not possible, then the alternative is to discard the infected program and reload a clean backup version.

The four generations of antivirus software:

1. First generation: simple scanners
2. Second generation: heuristic scanners
3. Third generation: activity traps
4. Fourth generation: full-featured protection

The advanced Antivirus Techniques are

- Generic Decryption
- Digital Immune System
- Behavior-Blocking Software

Generic Decryption: - Generic decryption (GD) technology enables the antivirus program to easily detect even the most complex polymorphic viruses, while maintaining fast scanning speeds. Recall that when a file containing a polymorphic virus is executed, the virus must decrypt itself to activate. In order to detect such a structure, executable files are run through a GD scanner, which contains the following elements:

- CPU emulator: A software-based virtual computer. Instructions in an executable file are interpreted by the emulator rather than executed on the underlying processor.
- Virus signature scanner: A module that scans the target code looking for known virus signatures.
- Emulation control module: Controls the execution of the target code.

Two major trends in Internet technology have had an increasing impact on the rate of virus propagation

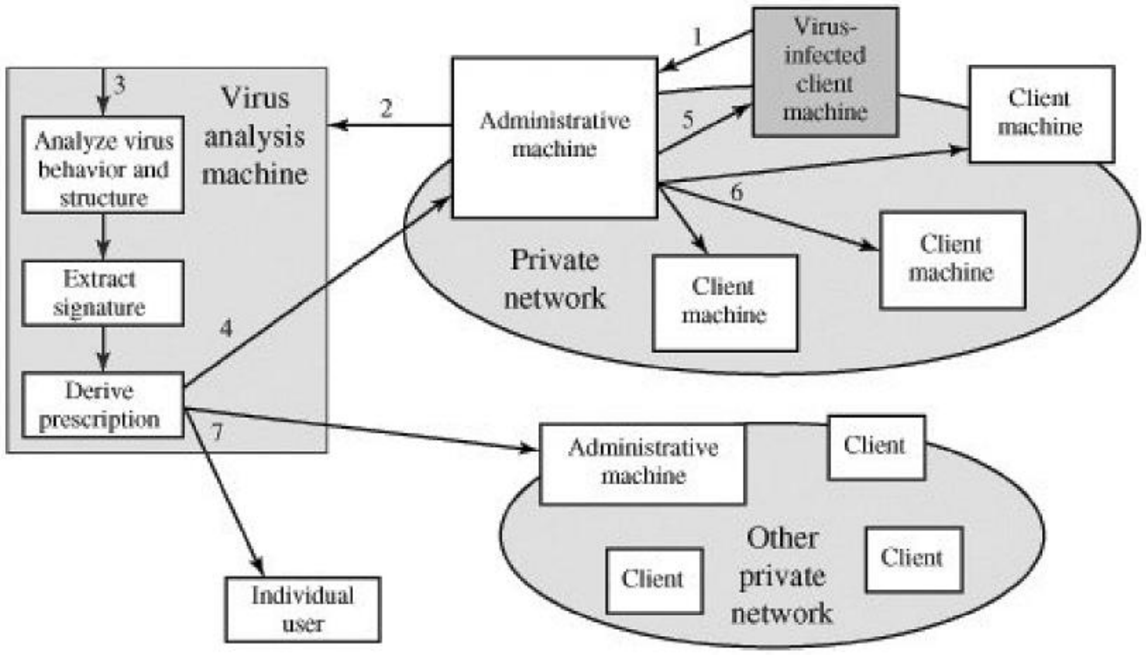
1. Integrated mail systems
2. Mobile-program systems

Digital Immune System: -The basic figure of a digital immune system is shown below. The Steps in digital immune system operation are

1. A monitoring program on each PC uses a variety of heuristics based on system behavior, suspicious changes to programs, or family signature to infer that a virus may be present.

The monitoring program forwards a copy of any program thought to be infected to an administrative machine within the organization.

2. The administrative machine encrypts the sample and sends it to a central virus analysis machine.
3. This machine creates an environment in which the infected program can be safely run for analysis. Techniques used for this purpose include emulation, or the creation of a protected environment within which the suspect program can be executed and monitored. The virus analysis machine then produces a prescription for identifying and removing the virus.
4. The resulting prescription is sent back to the administrative machine.
5. The administrative machine forwards the prescription to the infected client.
6. The prescription is also forwarded to other clients in the organization.
7. Subscribers around the world receive regular antivirus updates that protect them from the new virus.



Behavior-Blocking Software: - Behavior-blocking software integrates with the operating system of a host computer and monitors program behavior in real-time for malicious actions. The behavior blocking software blocks potentially malicious actions before they have a chance to affect the system. Monitored behaviors can include the following:

1. Attempts to open, view, delete, and/or modify files;
2. Attempts to format disk drives and other unrecoverable disk operations;
3. Modifications to the logic of executable files or macros;
4. Modification of critical system settings, such as start-up settings;
5. Scripting of e-mail and instant messaging clients to send executable content;
6. Initiation of network communications.

If the behavior blocker detects that a program is initiating would-be malicious behaviors as it runs, it can block these behaviors in real-time and/or terminate the offending software. This gives it a fundamental advantage over such established antivirus detection techniques as fingerprinting or heuristics.