CMR INSTITUTE OF TECHNOLOGY

USN										

Internal Assesment Test – II

Sub:	NETWORK SECURITY			Sec	A, B C & D			Code:	18EC821
Date:	15 / 04 / 2023	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	ECE

Scheme & Solution

Marks

1 In the RSA algorithm system, it is given that $p = 7, q = 11, e = 17$ *and* $m = 8$. Find the cipher text and decrypt to obtain the plain text $'m'$. 10

Given $p = 7; q = 11; e = 17$ *and* $m = 8$
W.k.t.

The private key is : $K_r = \{d, p, q\}$
The public key is : $K_u = \{e, n\}$

Step 1: Compute n: $n = (p).(q) = (7)(11) = 77$

Step 2: Compute $\emptyset(n)$: $\emptyset(n) = (p-1)(q-1)$
$= (6)(10)$
$= 60$

Step 3: Compute $gcd(e, \emptyset(n)) mod \emptyset\ (n) = gcd(17, 60) mod 60$
$= 1\ mod\ 60$; as expected

Step 4: Compute $'d'$ where $d.e = 1\ mod \emptyset(n)$
$d = e^{-1}\ mod\ \emptyset(n)$
$d = 17^{-1}\ mod\ 60$

Using extended Euclidean algorithm to find the inverse of 17 is as follows

q	r_1	r_2	r	t_1	t_2	$t = t_1 - qt_2$
3	60	17	9	0	1	-3
1	17	9	8	1	-3	4
1	9	8	1	-3	4	-7
8	8	1	0	4	-7	11
	1	0		-7	11	

$\therefore d = -7\ mod\ 60$
$\rightarrow d = 53\ mod\ 60$

Verification: $d.e = (17)(53) mod\ 60 = \frac{901}{60} mod\ 60$
$d.e \equiv 1\ mod\ 60$

∴ we have

The private key is : $K_r = \{d, p, q\} = \{53, 7, 11\}$
The public key is : $K_u = \{e, n\} = \{17, 77\}$

Encryption:
w.k.t. the cipher text 'C' is given as

$C = m^e\ mod\ n$
$= 8^{17}\ mod\ 77$

Using square and multiply algorithm we can compute $m^e = 8^{17}$ as follows

We have exponent $e = (17)_{10} = (10001)_2$; base $m = 8$

$1: \rightarrow 8 \; mod \; 77 = 8$

$0: \rightarrow 8^2 \; mod \; 77 = 64$

$0: \rightarrow 64^2 \; mod \; 77 = 15$

$0: \rightarrow 15^2 \; mod \; 77 = 71$

$0: \rightarrow (71^2) * 8 \; mod \; 77 = 57$

$$\therefore C = 57$$

05

Verification by decryption:

w.k.t. the plain text 'm' is given as

$$m = C^d \; mod \; n$$
$$= 57^{53} \; mod \; 77$$

Using square and multiply algorithm we can compute $C^d = 57^{53}$ as follows

We have exponent $d = (53)_{10} = (110101)_2$

$1: \rightarrow 57 \; mod \; 77 = 57$

$1: \rightarrow (57^2) * 57 \; mod \; 77 = 8$

$0: \rightarrow 8^2 \; mod \; 77 = 64$

$1: \rightarrow (64^2) * 57 \; mod \; 77 = 8$

$0: \rightarrow 8^2 \; mod \; 77 = 64$

$1: \rightarrow (64^2) * 57 \; mod \; 77 = 8$

$$\therefore m = 8$$

05

Hence verified

2 Explain the Diffie –Hellman key exchange algorithm. Also calculate the Y_A, Y_B and secret key (K_S) for q= 23, α = 7, X_A = 3 and X_B = 6. 10

Given $q = 23$; $\alpha = 7$; $X_A = 3$; $X_B = 6$

A's public key can be computed as: 02

$$Y_A = \alpha^{X_A} \; mod \; q$$
$$= 7^3 \; mod \; 23$$
$$Y_A = 21$$

B's public key can be computed as: 02

$$Y_B = \alpha^{X_B} \; mod \; q$$
$$= 7^6 \; mod \; 23$$
$$Y_B = 4$$

A computes the Shared secret Key as 03

$$K_{AB} = \alpha^{X_A X_B} \; mod \; q$$
$$K_{AB} = (\alpha^{X_B})^{X_A}$$
$$K_{AB} = (Y_B)^{X_A}$$
$$K_{AB} = (4)^3 \; mod \; 23$$
$$K_{AB} = 18$$

B computes the Shared secret Key as 03

$$K_{BA} = a^{X_A X_B} \bmod q$$
$$K_{BA} = (a^{X_A})^{X_B}$$
$$K_{BA} = (Y_A)^{X_B}$$
$$K_{BA} = (21)^6 \bmod 23$$
$$K_{BA} = 18$$

$$\therefore \textit{The seceret key is } K_s = 18$$

3 Explain the architecture of a distributed intrusion detection system. Give the major issues in the design. 10

Until recently, work on intrusion detection systems focused on single-system stand-alone facilities. The typical organization, however, needs to defend 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. **10**

The following major issues in the design of a distributed intrusion detection system

- **A distributed intrusion detection system may need to deal with different audit record formats. Ina heterogeneous environment, different systems will employ different native audit collection systems and, if using intrusion detection, may employ different formats for security-related audit records.**
- **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. Therefore, there is a requirement to assure the integrity and confidentiality of these data. Integrity is required to prevent an intruder from masking his or her activities by altering the transmitted audit information. Confidentiality is required because the transmitted audit information could be valuable.**
- **Either a centralized or decentralized architecture can be used. With a centralized architecture, there is a single central point of collection and analysis of all audit data. This eases the task of correlating incoming reports but creates a potential bottleneck and single point of failure. With a decentralized architecture, there are more than one analysis centers, but these must coordinate their activities and exchange information. A good example of a distributed intrusion detection system is one developed at the University of California at Davis [HEBE92, SNAP91]. Figure 18.2 shows the overall architecture, which consists of three main components:**
- **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.**

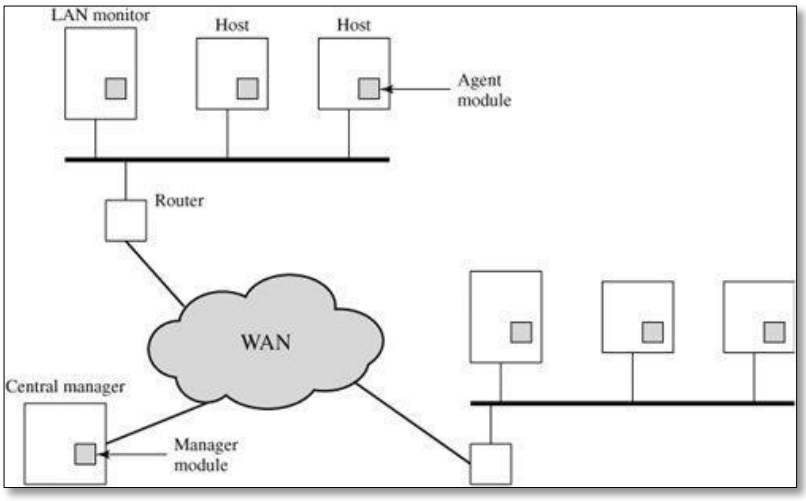

Fig 3a: Architecture for Distributed Intrusion Detection

The scheme is designed to be independent of any operating system or system auditing implementation. Figure 18.3 [SNAP91] shows the general approach that is taken. 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).

Next, 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.

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*. The architecture depicted in Figures 3a and 3b is quite general and flexible. It offers a foundation for a machine-independent approach that can expand from stand-alone intrusion detection to a system that is able to correlate activity from a number of sites and networks to detect suspicious activity that would otherwise remain undetected.

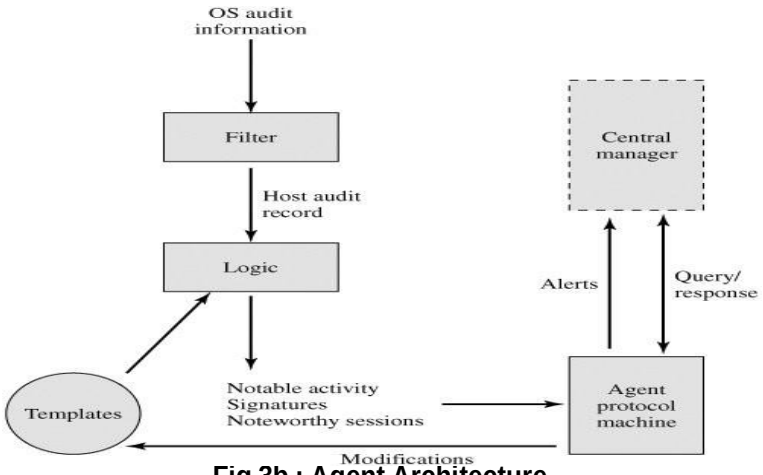

Fig 3b : Agent Architecture

4 Explain a detailed account on UNIX password management. 10

Password Protection **10**

The front line of defense against intruders is the password system. Virtually all multiuser systems require that a user provide not only a name or identifier (ID) but also a password. The password serves to authenticate the ID of the individual logging on to the system. In turn, the 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 "super user" 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.**

The Vulnerability of Passwords

To understand the nature of the threat to password-based systems, let us consider a scheme that is widely used on UNIX, in which passwords are never stored in the clear. Rather, the following procedure is employed (Figure 4a). 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.

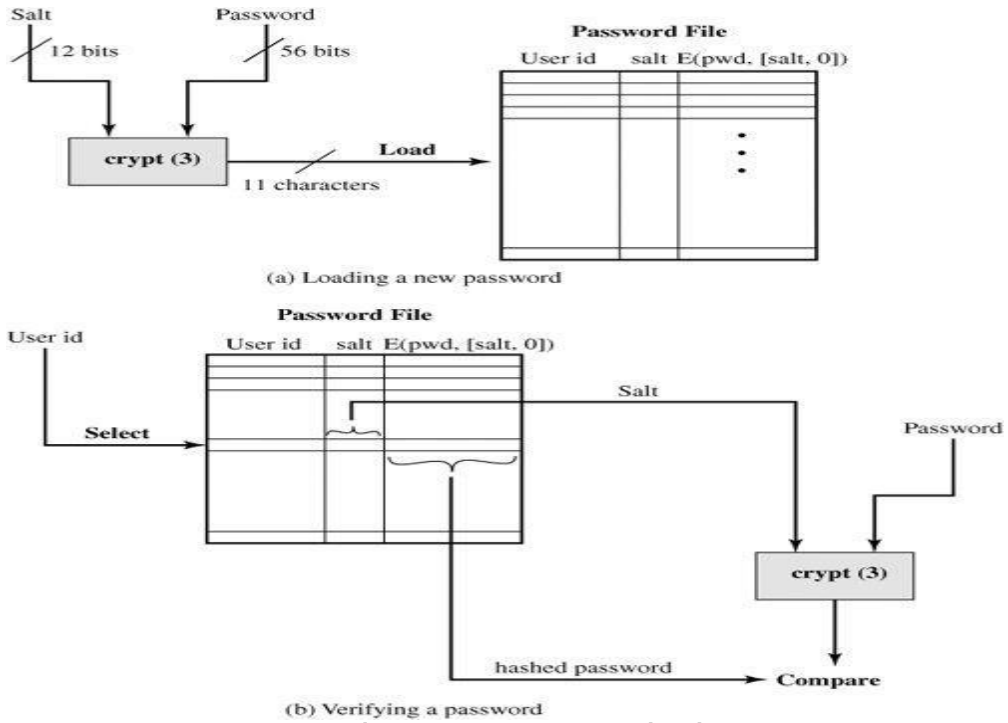

Figure 4. UNIX Password Scheme

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 password are used as input to the encryption routine. If the result matches the stored value, the password is accepted. The encryption routine is designed to discourage guessing attacks. Software implementations of DES are slow compared to hardware versions, and the use of 25

iterations multiplies the time required by 25.

However, since the original design of this algorithm, two changes have occurred. First, newer implementations of the algorithm itself have resulted in speedups. For example, the Internet worm described in Chapter 19 was able to do online password guessing of a few hundred passwords in a reasonably short time by using a more efficient encryption algorithm than the standard one stored on the UNIX systems that it attacked. Second, hardware performance continues to increase, so that any software algorithm executes more quickly.

Thus, 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.

As an example, a password cracker was reported on the Internet in August 1993 [MADS93]. Using a Thinking Machines Corporation parallel computer, a performance of 1560 encryptions per second per vector unit was achieved. With four vector units per processing node (a standard configuration), this works out to 800,000 encryptions per second on a 128-node machine (which is a modest size) and 6.4 million encryptions per second on a 1024-node machine.

Even these stupendous guessing rates do not yet make it feasible for an attacker to use a dumb bruteforce technique of trying all possible combinations of characters to discover a password. Instead, password crackers rely on the fact that some people choose easily guessable passwords.

5 Explain in brief the taxonomy of malicious programs. 10

Malicious software can be divided into two categories: those that need a host program, and those that are independent. The former are essentially fragments of programs that cannot exist independently of some actual application program, utility, or system program. Viruses, logic bombs, and backdoors are examples. The latter are self-contained programs that can be scheduled and run by the operating system. Worms and zombie programs are examples. **10**

We can also differentiate between those software threats that do not replicate and those that do. The former are programs or fragments of programs that are activated by a trigger. Examples are logic bombs, backdoors, and zombie programs. The latter consist of either a program fragment or an independent program that, when executed, may produce one or more copies of itself to be activated later on the same system or some other system. Viruses and worms are examples.

Trapdoor (Backdoor)

A backdoor, also known as a trapdoor, is a secret entry point into a program that allows

someone that is aware of the backdoor to gain access without going through the usual security access procedures. Programmers have used backdoors legitimately for many years 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.

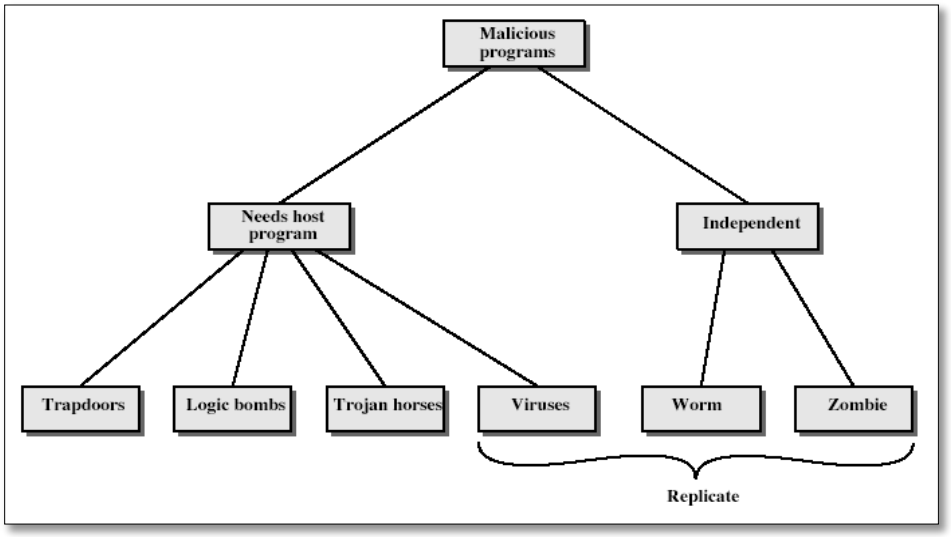

Figure 5 : Taxonomy of malicious software

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 backdoor 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.

Backdoors become threats when unscrupulous programmers use them to gain unauthorized access. The backdoor was the basic idea for the vulnerability portrayed in the movie *War Games*. Another example is that during the development of Multics, penetration tests were conducted by an Air Force "tiger team" (simulating adversaries). One tactic employed was to send a bogus operating system update to a site running Multics. The update contained a Trojan horse (described later) that could be activated by a backdoor and that allowed the tiger team to gain access. The threat was so well implemented that the

Multics developers could not find it, even after they were informed of its presence. It is difficult to implement operating system controls for backdoors. 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. Once triggered, a bomb may alter or delete data or entire files, cause a machine halt, or do some other damage.

A striking example of how logic bombs can be employed was the case of Tim Lloyd, who was convicted of setting a logic bomb that cost his employer, Omega Engineering, more than $10 million, derailed its corporate growth strategy, and eventually led to the layoff of 80 workers. Ultimately, Lloyd was sentenced to 41 months in prison and ordered to pay $2 million in restitution.

Trojan Horses

A Trojan horse is a useful, or apparently 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 files 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.

An example of a Trojan horse program that would be difficult to detect is a compiler that has been modified to insert additional code into certain programs as they are compiled, such as a system login program. The code creates a backdoor in the login program that permits the author to log on to the system using a special password. This Trojan horse can never be discovered by reading the source code of the login program.

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. For example, a CBS executive was victimized by a Trojan horse that destroyed all information contained in his computer's memory. The Trojan horse was implanted in a graphics routine offered on an electronic bulletin board system.

Zombie

A zombie is a program that secretly takes over another Internet-attached computer and then uses that computer to launch attacks that are difficult to trace to the zombie's creator. Zombies are used in denial of- service attacks, typically against targeted Web sites. The zombie is planted on hundreds of computers belonging to unsuspecting third parties, and then used to overwhelm the target Web site by launching an overwhelming onslaught of Internet traffic. Section 19.3 discusses zombies in the context of denial of service attacks.

The Nature of Viruses

A virus is a piece of software 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. Biological viruses are tiny scraps of genetic code DNA or RNA that can take over the machinery of a living cell and trick it into making thousands of flawless replicas of the original virus. Like its biological counterpart, a computer virus carries in its instructional code the recipe for making perfect copies of itself. The typical virus becomes embedded in a program on a computer. 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.

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 phases:

- 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.

Worms

A worm is a program that can replicate itself and send copies from computer to computer across network connections. Upon arrival, the worm may be activated to replicate and propagate again. In addition to propagation, the worm usually performs some unwanted function. An e-mail virus has some of the characteristics of a worm, because it propagates itself from system to system. However, we can still classify it as a virus because it

requires a human to move it forward. A worm actively seeks out more machines to infect and each machine that is infected serves as an automated launching pad for attacks on other machines.

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: a dormant phase, a propagation phase, a triggering phase, and an execution phase. The propagation phase generally performs the following functions:

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

2. Establish a connection with a remote system.

3. 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. In a multiprogramming system, it may also disguise its presence by naming itself as a system process or using some other name that may not be noticed by a system operator. As with viruses, network worms are difficult to counter.

6 With a neat diagram, explain digital immune system. 10

The digital immune system is a comprehensive approach to virus protection developed by IBM. The motivation for this development has been the rising threat of Internet-based virus propagation. We first say a few words about this threat and then summarize IBM's approach. **10**

Traditionally, the virus threat was characterized by the relatively slow spread of new viruses and new mutations. Antivirus software was typically updated on a monthly basis, and this has been sufficient to control the problem. Also traditionally, the Internet played a comparatively small role in the spread of viruses. But as [CHES97] points out,

two major trends in Internet technology have had an increasing impact on the rate of virus propagation in recent years:

- Integrated mail systems: Systems such as Lotus Notes and Microsoft Outlook make it very simple to send anything to anyone and to work with objects that are received.

- Mobile-program systems: Capabilities such as Java and ActiveX allow programs to move on their own from one system to another. In response to the threat posed by these Internet-based capabilities, IBM has developed a prototype digital immune system. This system expands on the use of program emulation discussed in the preceding subsection and provides a general-purpose emulation and virus-detection system.

The objective of this system is to provide rapid response time so that viruses can be stamped out almost as soon as they are introduced. When a new virus enters an organization, the immune system automatically captures it, analyzes it, adds detection and shielding for it, removes it, and passes information about that virus to systems running IBM AntiVirus so that it can be detected before it is allowed to run elsewhere.

Figure 6 illustrates the typical steps in digital immune system operation:
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.

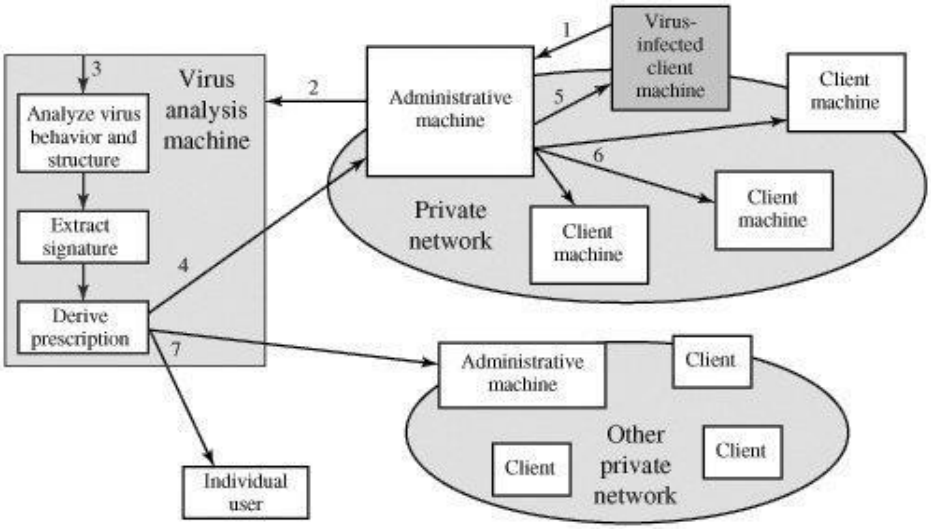

Figure 6. Digital Immune System