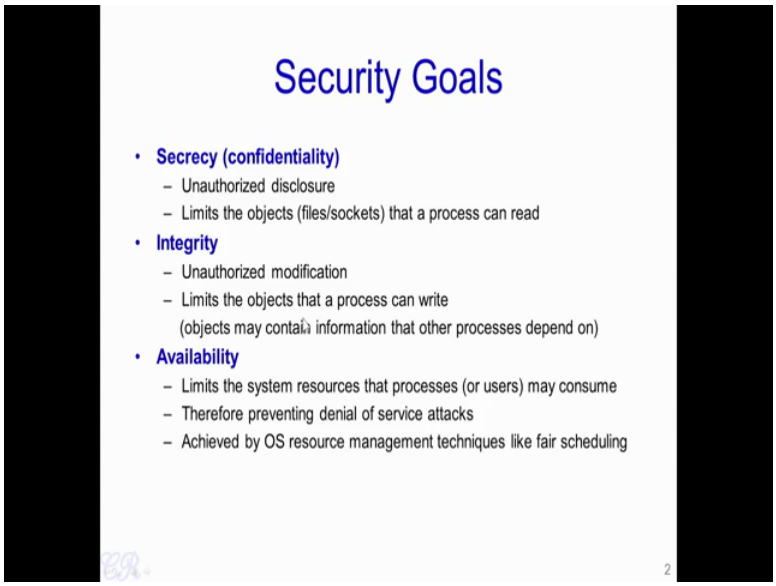
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 08
Lecture – 35
Operating System Security

Hello. In this video we will look at Operating System Security. Now, security as such is become extremely important in this current world because systems as such are always connected, as a result it becomes very easy for malicious programs to enter into the system. Therefore, operating systems should be designed in such a way so as to prevent as much as possible any loss of information due to such malicious intrusions.

So, we will see a brief introduction to Operating Systems Security in today's video.

(Refer Slide Time: 01:00)



So, whenever we design a system, we need to have some security goals and these goals are the Secrecy, Integrity and Availability of the system (mentioned in above slide image). Now, secrecy or confidentiality means that certain objects should not be visible to particular processes. For instance depending on the privilege of a process and also on the privilege of the user running that process, certain objects like certain files stored in the desk or certain network sockets should not be readable by that process. Essentially what confidentiality achieves is that unauthorized disclosure of some objects should be

prevented. What secrecy or confidentiality achieves is that unauthorized disclosure of certain objects like files and sockets should not happen.

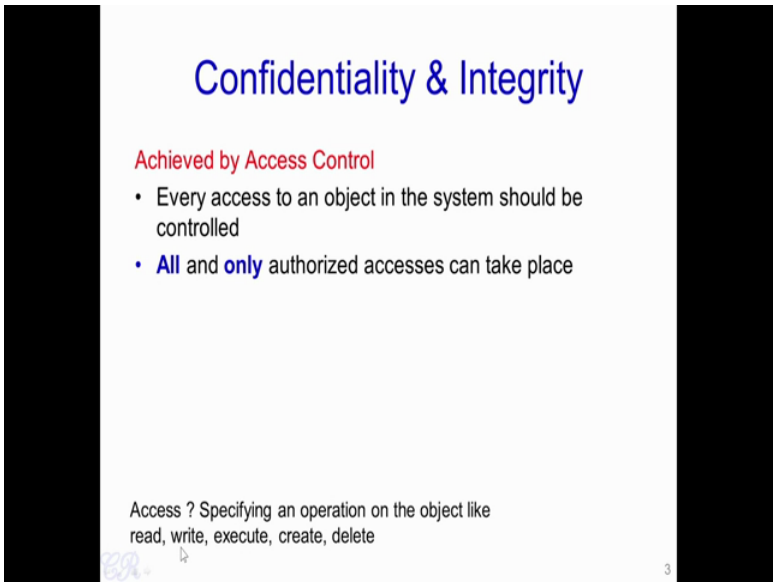
With respect to integrity means unauthorized modifications. In other words a malicious user or a user with not sufficient privileges should not be able to write to a particular object, such as a file or a socket. To take an example, a normal user in the system should not be able to modify some system related files.

Availability, on the other hand means that a particular user or a particular process has a limited use of the system resources. In other words, one particular user should not be able to hog the entire system resources.

If such a thing (Availability) is not available in the system it could be lead to what is known as denial of service attacks, wherein some malicious programs running on the system will prevent any other program which is present from having access to certain hardware resources like the CPU, the RAM or disk. So, availability in a system largely depends on how the operating system gets designed.

For instance, we had seen how CPU schedulers can be designed in order to ensure fair scheduling that is every process in the system gets a fair share of the CPU depending on its priority. Now, in order to achieve secrecy and integrity, what operating systems generally do is to have access control. So, we will be looking at access control techniques in this particular video.

(Refer Slide Time: 04:04)



In order to achieve access control, what it means is that every access to an object such as a file, a socket or it could be a hardware device like a printer or a monitor should be controlled. Essentially, if a process wants to get access to a particular file or any other object in the system, it has to go through an access control mechanism. Essentially, all and only authorized accesses can take place. In other words, if a process needs to access an object then it definitely should be given access to that particular object. So, what we mean by access? Essentially by access we mean an operation on the object which a particular process intends to do.

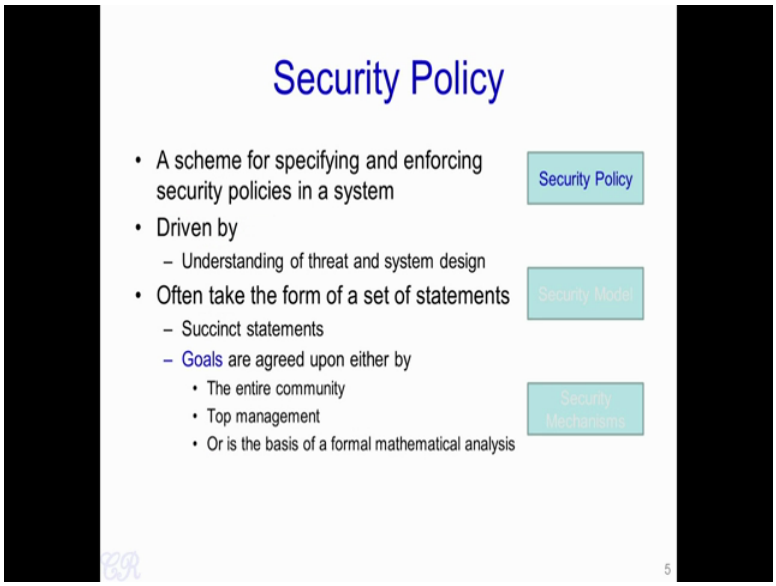
So, these operations could be one of read, write, execute, create or delete. Now, this access should be permitted if and only if the particular process is authorized to make that access. In other words, only if a process is authorized to execute a particular program or read or write to a particular file only then should the operation be permitted.

(Refer Slide Time: 05:39)

In order to develop an Access Control System, there are three components: Security policies, Security model and Security mechanism (mentioned in above slide image). Now security policy, are high level rules that define the access control. Now security model is a formal representation of the access control security policy and its working. Essentially this (security model) is a formal model or a mathematically represented model of the security policy. And this (security) model is used to help in proving various things about the system, for instance you could use this particular security model to prove that a particular system is secure.

Now, security mechanism is low level hardware or software that has the functional implementation of the policy that is the security policy and the model. So, in other words you can think of this (i.e security policy) as the high level rules that define the security policies or the security features that need to be supported by the system. This (i.e Security Model) is a mathematical model or slightly lower level representation and while this mechanism is the actual implementation of the policies. So, let us look at each of these more in detail.

(Refer Slide Time: 07:10)



So, Security policy is essentially a scheme for enforcing some policies within the system. So how do we decide upon what policies a system should have? This is defined by what threats are present in the system as well as how the system is designed. So, developing a security policy is not easy. So it would require a lot of brain storming to actually identify what are the various threats that or what are the various attacks that are possible on the system and once these attacks are known then policies are created in order to prevent such attacks. So, how is the security policy actually look like?

So a security policy essentially would be a set of statements. These statements should be extremely succinct and precise and should have the goals of the policy. So, these goals should be agreed upon by either the entire community or the entire set of people who are developing that system, or by certain top management and this forms the basis for the security model which is the formal mathematical representation of the policy.

(Refer Slide Time: 08:42)

So let us take how a security policy should not be written. So, what we are seeing now is that security policies are not necessarily for computer systems, but it could also be for organizations (as mentioned in above slide image). So, this particular security policy is for this particular company called Megacorp Incorporated. So, there security policy is extremely short, it has just four statements and it reads as follows; this policy is approved by the management, all staff shall obey the security policy, data shall be available only to those with a 'need-to-know', all breaches of this policy shall be reported at once to security.

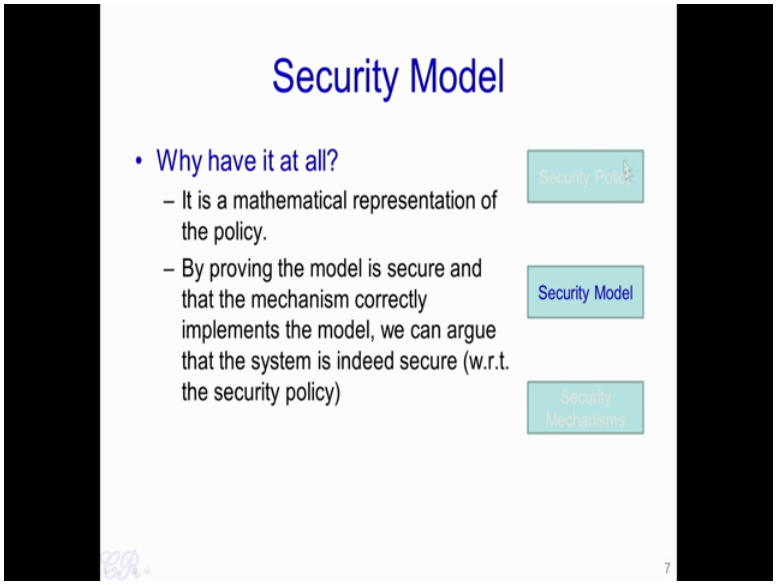
So what is the issues with this particular security policy? So if we actually take a more closer look at this policy from this particular company, what we see that there are lot of flaws in this. So, essentially the policy is not complete and cannot be used to build a security mechanism. So for instance, this policy is approved by management the first thing (i.e first policy mentioned in above slide image). So, typically in our security policy the approval of the policy should not be part of the policy document itself.

Second, all staff shall obey (mentioned in above slide). So, what we mean by this? Who enforces that a staff should obey? Is it moral a requirement of the staff to actually obey to this policy or there is actually a unit which enforces all staff in the organization to obey for this particular policy? Third, the 'need-to-know', so how or how to be defined what I need to know? Who determines who should know or what information? So, this is not

specified or not very obvious from this particular policy statement.

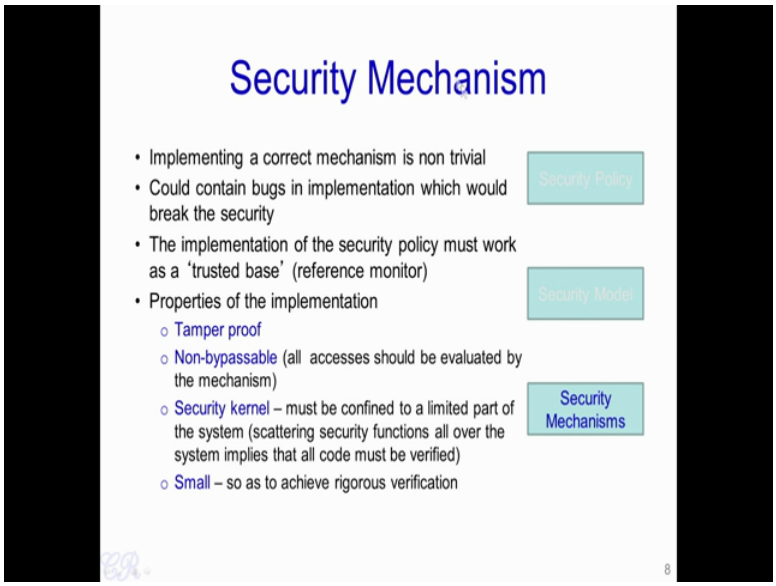
Finally, all breaches of this policy should be reported at once the security. So, how are breaches deducted? The security policy does not tell anything of this form? Whose duty is to report them? So this two is not mentioned over here (in above security policies), so as you see even though this particular policy is very short and very tars and incorporates just four points, still it leads to a lot of ambiguity and such ambiguity would lead to a weak security for that company. So, when we have to write a security policy for a company it should be complete in all aspects.

(Refer Slide Time: 11:36)



Now, let us look at the security model (mentioned in above slide image). Essentially why do we need to have the security model at all? Why cannot we go from the security policy, use the security policy and directly implement security mechanisms? Why do we need to have a security model present in this entire hierarchy? Essentially by having a security model, we are able to model the security policy in a lower and more formal construct. So in this way any gaps in the security policy would be deducted and secondly, we could also use tools and techniques in order to argue or prove that the system is indeed secure with respect to the security policy that was defined.

(Refer Slide Time: 12:31)



So, the last aspect is the Security Mechanism (as mentioned in above slide image). So Security Mechanism as we have seen deals with implementing of the security policy. Now, it is important that implementing the security mechanism is bug free, it has no bugs. Why is it extremely important that there are no bugs? This is because if there are bugs in the implementation of the mechanism then it would be possible for attackers to exploit that bugs and get in unauthorized accesses into that system.

Second, the implementation of the security mechanism should be the trusted base. Essentially this is the core from where the security of the entire system would depend on. Therefore, if it itself is buggy and incomplete, it will not create a secure system. So, properties of the security mechanism implementation is that it should be tamper proof (1st property), non-by passable (2nd property). And by non-by passable, we mean that all accesses into the system should first be evaluated by the security mechanism. Then it should be a security kernel (3rd security kernel) so, what it means by this is that the entire implementation for the security policy should be confined to a limited part of the system and it should be scattered into various parts of the system.

So, having a security kernel where everything is just confined into, say a small part of the entire system would make it easy to test and debug and verify in terms of security. Again it should also be small (4th property). So, a small size for the security mechanism will ensure that it can undergo some rigorous verification.

(Refer Slide Time: 14:35)

Now, there are 3 techniques for access control. They are the DAC, MAC and RBAC. So, DAC is the Discretionary Access Control, MAC is Mandatory Access Control and RBAC is Role-based Access Control. So, we will be looking at the DAC and MAC in this particular lecture.

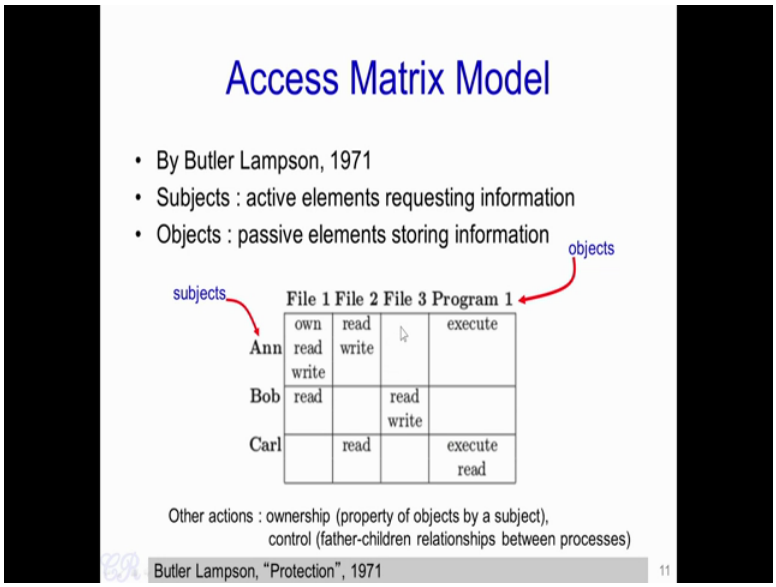
(Refer Slide Time: 15:00)

Discretionary Access Control or DAC (mentioned in above slide image) is an access control which is based on the identity of the requester. There would be a set of access rule what requesters are (or not) allowed to do. So, these access rules would define

essentially what objects the requester, it also known as the subject could access and what objects could be read, written to, executed, created or deleted and so on.

Now, the privileges for these various objects are granted as well as revoked by the system administrator. So, users if they have a particular privilege then they can also pass on that privilege to other users. A very common example of a DAC system that is the discretionary access control system is the Access Matrix Model.

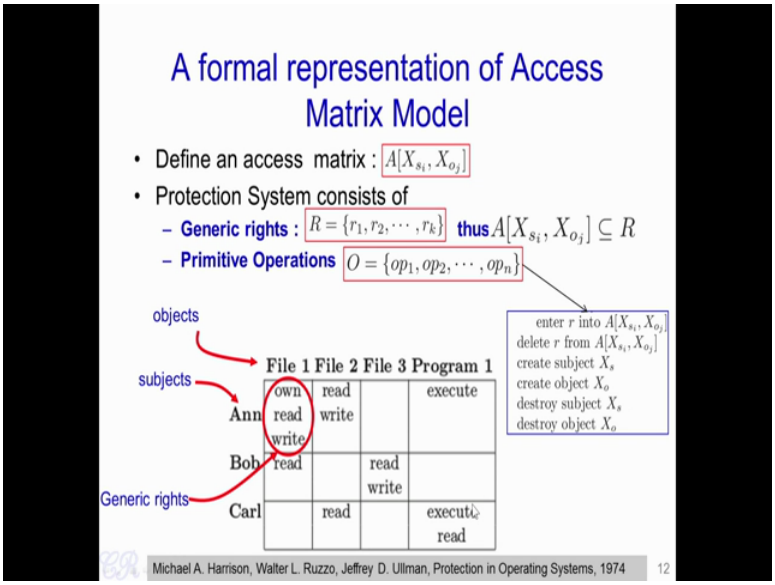
(Refer Slide Time: 16:05)



So, the Access Matrix Model was designed by Butler Lampson in 1971. It comprises of subjects and objects, in addition to this it also has a table in this form (mentioned in above slide image) where each cell has some particular actions. So, subjects are active elements in the system, such as users of the systems or processes or programs that are requesting information.

Objects on the other hand are passive elements which for instance store information. Now the actions are specified in each cell of this particular matrix. So, for instance the subject Ann has execute permissions for program 1, she has read and write permissions for file 2 and she has read and write permission for file 1 as well as she is the owner for file 1. So, if you look more carefully, Ann does not have any permission on file 3, she can neither read it nor write it nor she can actually read or write the program 1 (mentioned in above slide image).

(Refer Slide Time: 17:27)



So, in order to represent this access matrix model in a formal method, what we can do is define this matrix A which comprises of X_{si}, that is for subjects and X_{oj}, which is for an object (mentioned in above slide image). So, A of this particular thing (i.e $A[X_{si},X_{oj}]$) would indicate the subject si, for instance Bob and an object for instance file 2 (refer above slide image). Then we could also give rights to each cell in this particular matrix.

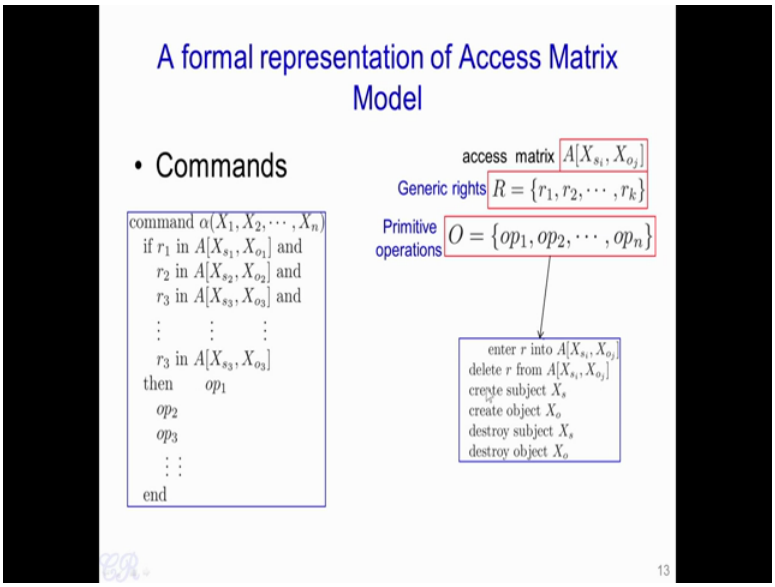
For example, the rights could be taken from a set like this $R= \{r_1, r_2, \ldots, r_k\}$ and this particular thing $A[X_{si},X_{oj}]$ is a subset of R. So for example, the rights in this example (as mentioned in above slide image) is own, read, write, execute, so on and these would be in the set R and corresponding to each cell in this matrix we define what the rights are. So, this would define what the right for subject si has on the object oj. In addition to this there is something known as the primitive operations. So, this is represented by O and it is a set of 6 operations which is specified here (mentioned in box in above slide image).

So this is for instance, enter r into some location in the matrix or delete r from that location, here r is taken from the generic rights (as mentioned in above slide image). So, it is an element form the set R. Similarly other operations are to create a particular subject X_s, create an object X_o, destroy a subject X_s, or destroy an object X_o (refer slide time 17:27).

So these are the only primitive operations that can be done on this particular access matrix (as mentioned in above slide). So, as we see each of these primitive operations

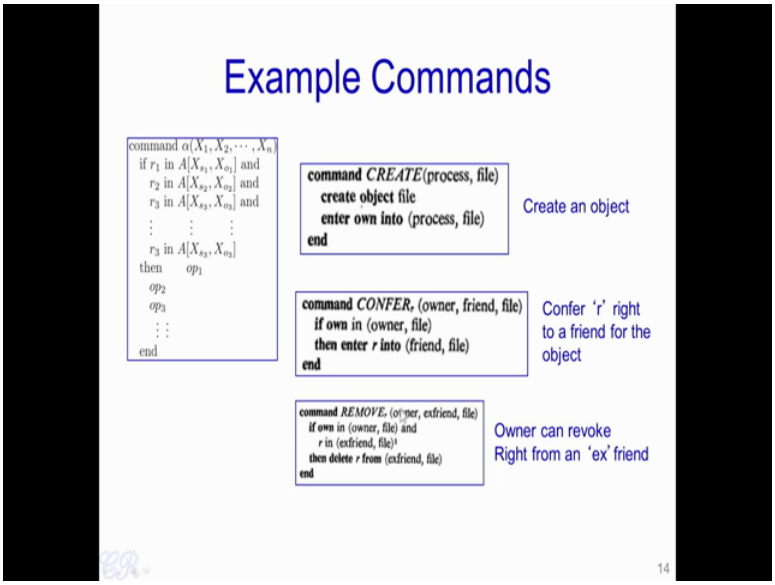
would either modify the contents of a particular cell in the matrix or it could delete an entire column with the destroy subject and destroy object. So it could either delete a particular column or a particular row or if it wants it could also create a new column that is creating an object or creating a subject. So once, we have defined such primitive operations, we can define something which is a bit more complex and paste on these primitive operations which are known as Commands.

(Refer Slide Time: 20:17)



So, to recollect we have the access matrix which is specified by A (i.e $A[X_{si},X_{oj}]$), we have the generic rights r_1 to r_k (i.e $R= \{r_1, r_2, \ldots, r_k\}$) and we have 6 primitive operations as we specify here (mentioned in above slide image in box). So, we can use all of this to create complex commands like this way (mentioned in above slide image). A typical command would look like this $\alpha(X_1, X_2, \ldots, X_n)$ and it would be called say, for instance α which takes several parameters X_1 to X_n and we can have a set of rules. For instance if r_1 is in this matrix cell (i.e $A[X_{si}, X_{oj})]$ and r_2, r_3 and so on, then perform these following operations (i.e op_1, op_2 etc). So, what we are saying is that based on these low level primitive operations, the genetic rights and the access matrix, we can have more complicated commands on the access matrix.

(Refer Slide Time: 21:19)

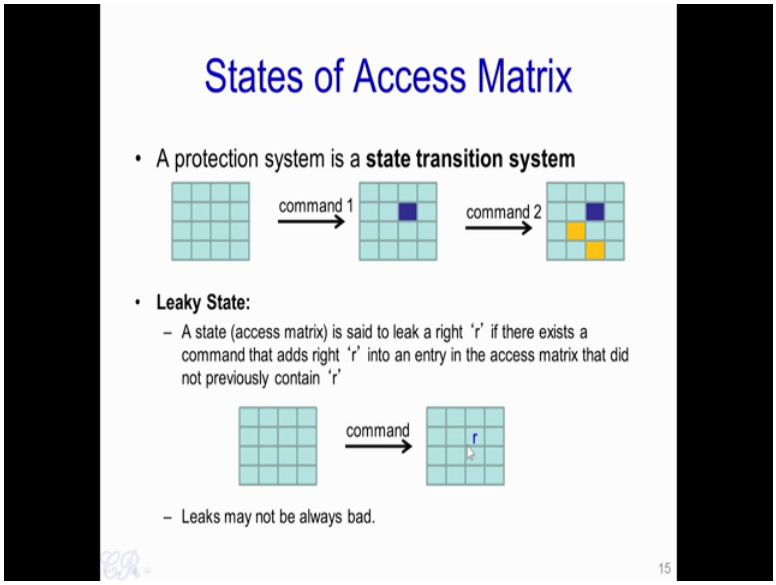


So let us take a few examples of such commands, for instance CREATE(process, file). So, process here is the subject and what it means to say is, this process (first argument) wants to create this file (second argument). So, the command will work like this (first command mentioned in above slide image), so first we use the primitive operations, create an object file and then enter own into (process, file). So, this process, file corresponds to the cell in the access matrix. So if this particular command is not present, then it will not be possible for a process to actually create a particular file. So, this is a very simple example.

We could have something a bit more complicated like this particular example that is $CONFER_r$ - owner, friend, file. So, owner and friend are the subjects and file is the object over here (second command mentioned in above slide image). And what is intended to do by this particular command on the access matrix is that, we want to confer right 'r' which is present over here (i.e $CONFER_r$) to a friend for that particular object. So, this owner (1^{st} arg) wants to confer the right 'r' for the friend (2^{nd} arg) with respect to this particular file (3^{rd} arg). So, what goes on in this particular command? So essentially we first test if the owner is the indeed the owner of that file. So, we see that first if own right is present in owner, file in the matrix cell corresponding to owner the subject and file the object (i.e own in (owner, file)), if this is indeed true then enter r that is enter the right 'r' into the friend, file part of the matrix (i.e enter r into (friend, file)).

Similarly, we can have another example, where we can remove a right from an exfriend. We are passing this particular command, three parameters: the owner, ex friend and the file (i.e $REVOKE_r$ (owner, exfriend, file). So, essentially what we want to do is revoke the particular right 'r' from this exfriend which is another subject. So, the command looks like this, if own is in the (owner, file) that is if owner is indeed the owner of the file and the right 'r' is present in (exfriend, file) then delete 'r' from (exfriend, file). Therefore, we can remove the right 'r' which could be a read or write or execute right, on the file with respect to exfriend. Essentially the exfriend subject will not have the right 'r' on the particular file.

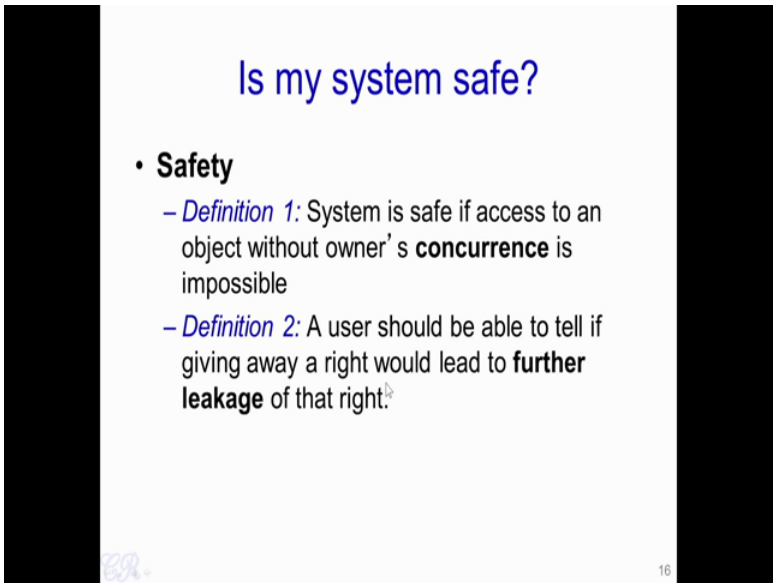
(Refer Slide Time: 24:24)



So a protection system, if we actually represent it by this particular matrix (first matrix mentioned in above slide image) and what we seen is that we could apply commands to it like we have seen a couple of commands in the examples in the previous slide (refer slide time 21:19) and with each command we are modifying this particular access matrix. So, this command for instance (i.e command 1) has modified this cell over here in the access matrix (second matrix mentioned in image). Similarly, if you run another command (I,e command 2), another part of the access matrix or multiple cells in the access matrix will be modified (third matrix mentioned above). So, based on this we define, what is known as a Leaky state? A state of the access matrix is said to leak right 'r' if there exists a command that adds the right 'r' into an entry in the access matrix that did not previously contain 'r'.

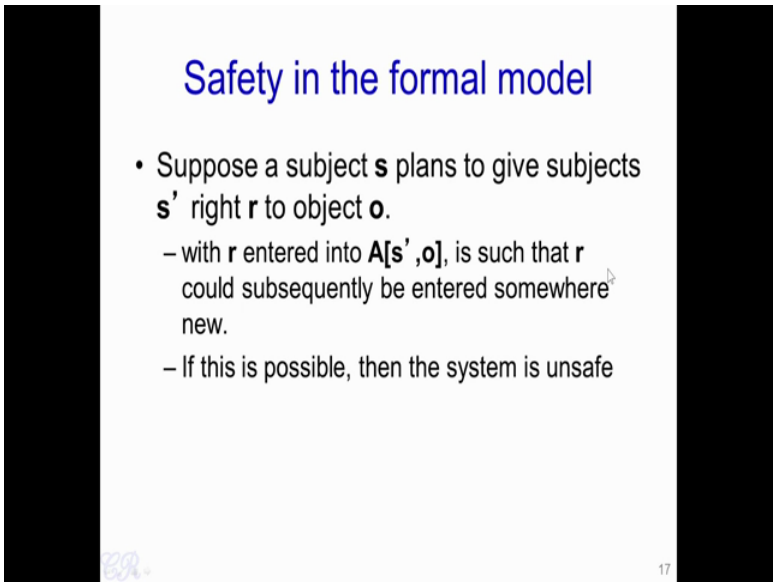
So, what this means is that we have the state of the matrix here (two matrix mentioned in leaky state section) and we run a command and as a result there is an r or right 'r' which is entered into this particular matrix (i.e second matrix). So, what we say is that this particular state of the matrix leaks r. Now, a leaky state does not always need to be bad. So, it is left to us to actually determine whether leaking this 'r' from a particular cell in the matrix is good or bad with respect to security of the system.

(Refer Slide Time: 26:03)



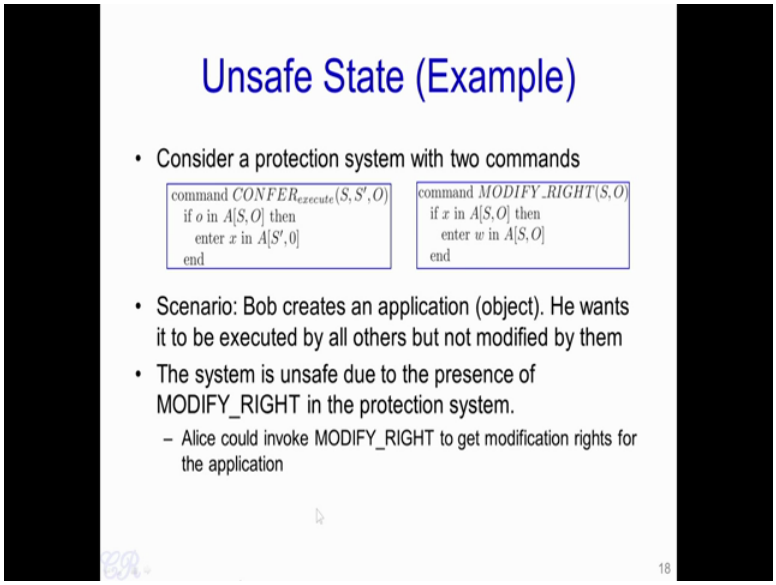
So let us define when a system is said to be safe? So, we have 2 definitions of safety. The 1st definition; a system is safe if access to an object without the owner's concurrence is impossible. 2nd, a user should be able to tell if giving away a right to a particular subject and with respect to an object would lead to further leakage of that right.

(Refer Slide Time: 26:43)



So let us see what this means form a formal model. Suppose, a subject 's' plans to give another subject s prime i.e s', the right r to object o. With r entered in A[s', o],that is with a new right added to this particular cell of the matrix. Is it possible that r could subsequently be entered somewhere else in the matrix A, if such a thing is possible then the system is set to be unsafe. So, essentially a system is unsafe if any operation or any command run on the access matrix could result in that particular right being conferred or transferred to someone else or to somewhere else in the matrix.

(Refer Slide Time: 27:47)



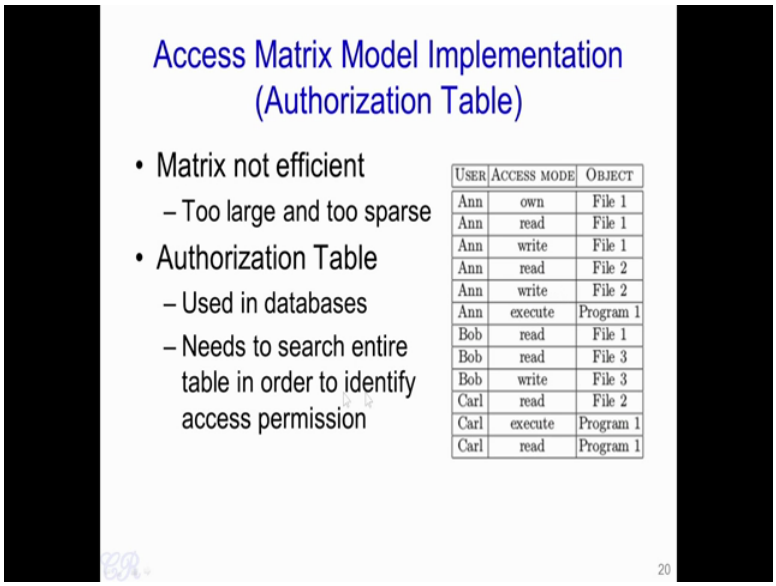
So we will look at an example of an unsafe state. So let us consider these two commands (mentioned in above slide image), one is the $CONFER_{execute}$. So, the subject S wants to confer the subject S prime, the execute right for O. So, this command $CONFER_{execute}$ (S, S', O) states that a subject if it is the owner of the object O then it can give the right 'x' that it can give the right to execute object O to another subject S prime (i.e x in A[S', O]).

Now, let us say our system also has this particular command which is called MODIFY_RIGHT(S, O) which states as follows (mentioned in above slide image), if a particular subject has an execute right 'x' with respect to an object then it can also enter a write in that particular object (i.e w in A[S, O]). In other words, what this MODIFY_RIGHT allows us to do is that if a particular subject can execute an object then it can actually modify its rights in order to change the object as well. So, essentially it can write into that object.

So let us look at a particular scenario to see, how this example shows an unsafe state (mentioned in above slide image). So, let us say, Bob creates an application object, he wants it to be executed by all others but not modified by them. So, the system is obviously unsafe due to the presence of the MODIFY_RIGHT in the protection system. Alice, another person who has the execute right for that particular object or application could invoke MODIFY_RIGHT to get the modification rights on that application and once Alice gets the right to modify that application then there is nothing stopping Alice, from actually changing that operation.

So, what we see is that because Bob has given the execute permission on a particular object, and the systems policies or the systems access control mechanism is built in such a way that because this right i.e 'x' is present in a particular object, then the w write is also given to that particular object. Essentially if a subject has the execute right for an object then that right is actually transformed and transferred and it will cause the w write to be also associated with that particular object with respect to that subject. Therefore, this particular state in the system is an unsafe state.

(Refer Slide Time: 30:48)

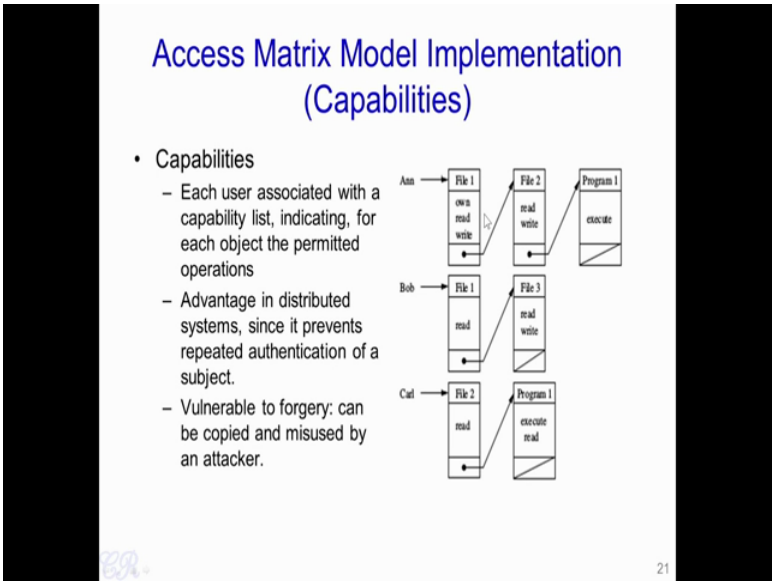


So how do we actually implement the access matrix model? So, one way as we have seen is to have the Matrix representation. However, the issue with this particular representation is that it is too large as well as it is too sparse. So, it is not a very efficient way to use.

The other technique is to use something known as an Authorization table as is shown over here (mentioned in above slide image). So this particular table is used in data bases and what it actually have is that each row comprises of a user, it has a right and it has an object. So, it says that for instance, Ann has a read, write on the object file 1 (mentioned in above image).

So, for every right that Ann holds there will be a separate row in this particular table. So, the problem with this particular scenario is that or this particular implementation (authorization table implementation) is that, one needs to at least search the entire table in order to identify whether a particular user has a particular right on a particular object. So, it again is having a lot of performance issues with respect to how long it takes to determine a particular right for a subject and object.

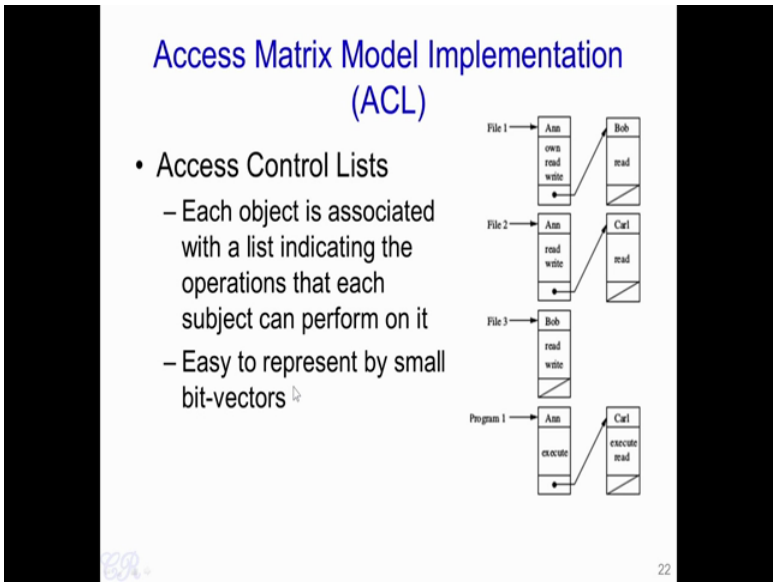
(Refer Slide Time: 32:14)



So another technique of implementing the access matrix model is by using Capabilities. Here (as mentioned in above slide image), each user is associated with a capability list, indicating, for each object the permitted operations. For instance, Ann, for file 1 has the following rights over here, for file 2 she has these rights, for program 1 she has these rights (refer above slide image). Similarly, for every user we have such a list and these are the capabilities list for that user with respect to each of these objects.

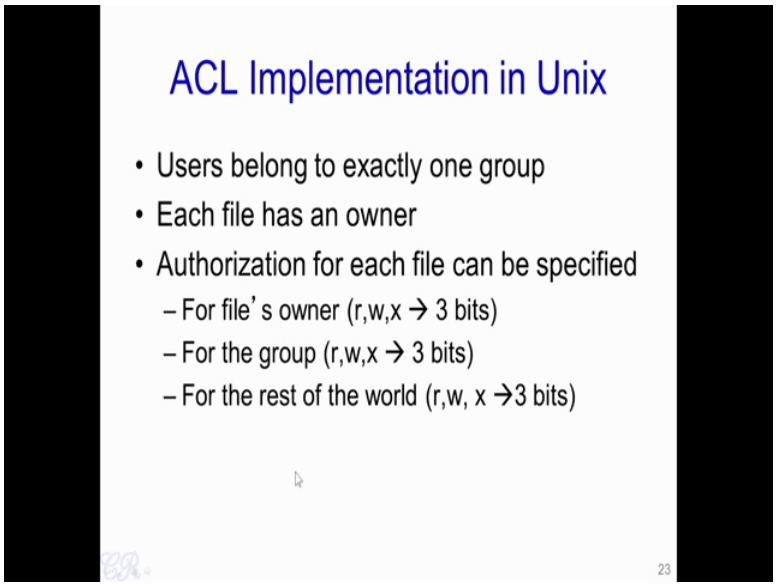
So the advantage of such a model or such an implementation is from a distributed system scenario, since it prevents repeated authentication of a subject. So, essentially once the subject authenticates itself in the distributed environment, then it could obtain this entire list and the system will know its rights for each object which is present. However, the limitation is that it is vulnerable to forgery if a particular right or a list is copied by an attacker, it can be then misused.

(Refer Slide Time: 33:34)



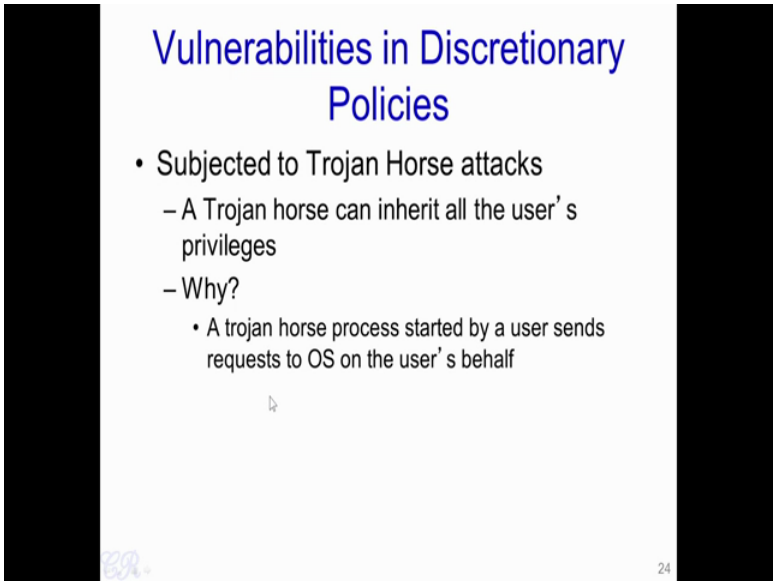
The third way of implementing the access control model is by what is known as ACL or Access Control List (mentioned in above slide image). Essentially, each object here is associated with a list indicating the operations that each subject can perform on it. Essentially, this is the opposite of the previous way. So, corresponding to each object we have a list and each node in the list has a subject and the corresponding operations that are present here (i.e read, write, execute). So, the advantage that we get with this particular thing is that it is easy to represent by small bit factors. So, if you look at how UNIX actually implements this ACL.

(Refer Slide Time: 34:19)



So, we see that every file in a UNIX system has 9 bits associated with it (mentioned in above slide image). So, this gives the authorization for each file to be specified. For instance there are 3 bits which specify read, write, execute for the file, for the owner of the file. Then we have the group for we have the 3 bits r, w, x for the group and for the rest of the world, for the others that is we have 3 more bits (authorization for each file as mentioned in above slide image). Thus in a UNIX system which uses the ACL implementation of the access matrix, there are 9 bits involved and each of these bits would give permissions to various users of that file.

(Refer Slide Time: 35:13)

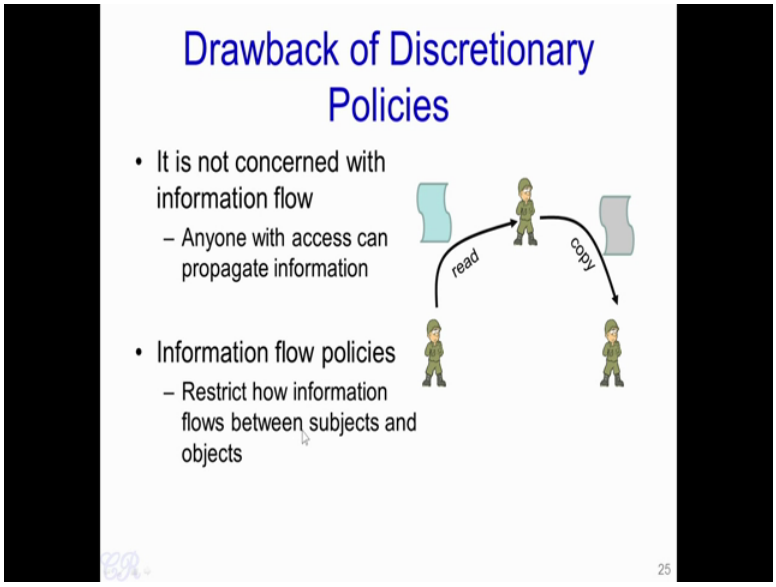


Now, the vulnerabilities in the discretionary policy is that, it is subjected to the Trojan Horse attacks. So what we mean by Trojan horse is that it is a small code which is present in a larger non-malicious code. So, the small Trojan present in the larger code is the malicious code. So the issue with this is that because the Trojan is actually like a virus which is joined with a much larger program, so the Trojan can actually inherit all the users or all the processes privileges.

So for instance if you have a Trojan which is connected to let say a program which has a super user privileges to execute or which needs to execute as a route, then the Trojan itself would be able to inherit all the route permissions. The reason why this happens is that the Trojan present in the process would send request to the operating system in the valid user's behalf. So from the OS perspective, the operating system will not be able to

distinguish whether it is from the valid process, which is started by a valid user or whether the request is coming from a Trojan. Therefore, it becomes difficult to detect.

(Refer Slide Time: 36:51)



Another drawback of discretionary policy is that it is not concerned with information flow; anyone with access to information can also propagate information. For instance if we have this one person over here (mentioned in above slide image in middle) who is able read a particular file, there is nothing stopping this person from actually making a copy of this file and transferring it to hundred different people. So as such the discretionary policies do not take care of the flow of information from one person or one user to another. So it is only capable of preventing access to information, but not the flow of information.

On the other hand, in order to consider flow of information also, we need to use what is known as information flow policies which restrict how information flows between subjects and objects.

So, we will look at Information Flow Policies in the next video.

Thank you.

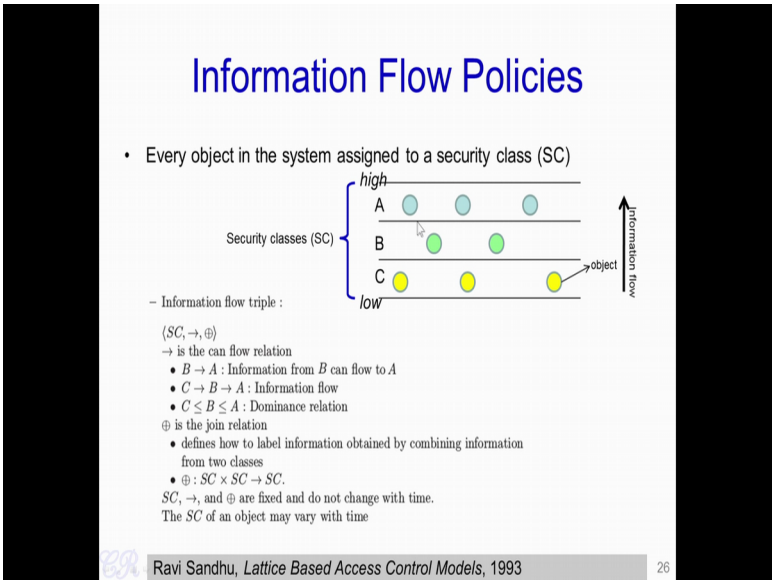
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 08
Lecture - 36
Information Flow Policies

Hello. In the previous video we had looked about Access Control Techniques, and in particular we had looked into the DAC or the Discretionary Access Control. So, we had seen that DAC is very efficient in giving certain access rights to a particular subject with respect to a set of objects. However, what we seen was the drawback of DAC was that it was incapable of preventing information flow.

So, in this video, what we are going to see is information flow techniques and the MAC that is the Mandatory Access Control.

(Refer Slide Time: 01:03)



In Information Flow Policies, every object in the system is assigned to a particular security class. As such the system is divided into a fixed number of security classes and each security class is then given a particular category.

For example you have a security class which is high, a security class which is low, and so on. Now each object in the system is assigned to one of the security class. Similarly each subject in the system that is the entities which actually operate or access these objects they also are given a particular security class (mentioned in above slide image). Next we will define how information flows between the classes.

Note that here about information flow, we are not concerned about how information about one object flows to another object or flows to a subject. But rather a concern is more about classes. So, what we are going to see is that how information from one class flows to another class.

So, formally this is represented by this following triple (refer slide time 01:03). So, this triple is a 3 triple, it contains the SC that is a security class. It has this arrow operator which shows the flow relation, and it has the join operator which is the join relation. So, an operator like this B -> A implies that information from B can flow to A. Similarly this example C arrow B to arrow A shows that information from C flows to B, and information from B flows to A. In other words information from C can also flow to A (mentioned in above slide image). Now based on this there is another technique of representing, which is by this particular symbol the <= symbol. So, what it means is that A dominates B and B dominates C.

Now, let us look at the join relation (refer slide time 01:03). So, essentially the join relation is used to determine how to label information, after information from 2 classes is combined. So, for instance, let us say we are creating a new object say a new file, and this file is for instance created by concatenating an object present in the security class C and an object in security class B.

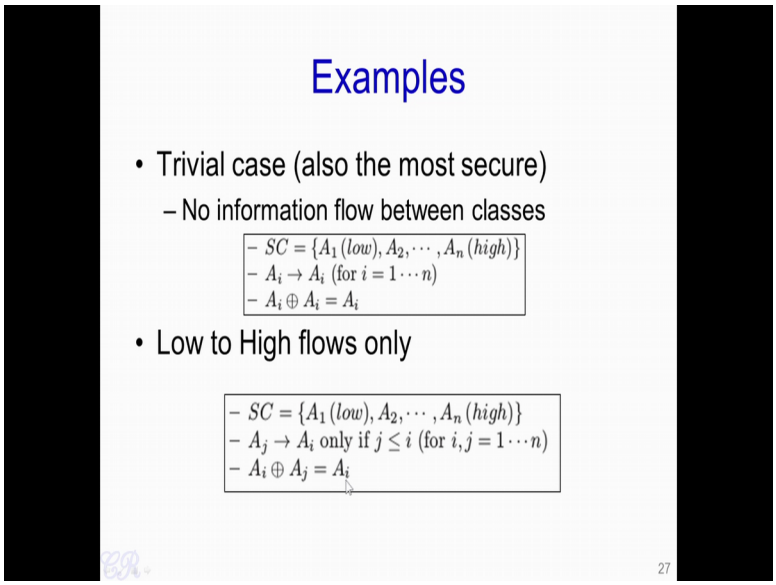
So now, the question that arises is to which class should the new file that we have just created be present in, should it be in security class C or security class B or in the A security class? So, in order to formally write this, the join operator is used (mentioned in above slide image). So, join is defined as a function between 2 classes. So, it takes the security class and another security class and it will actually give us what is the resulting security class i.e SC X SC -> SC.

So, we will see and understand more about this with an example. Now what we see is that for the system, these security classes are fixed. So, during the design time itself for instance we would say that our system has for instance just the security class A, B and C, also the flow operations is fixed at the design time itself. So, we can design various scenarios saying that information can flow to C and from C to B and from B to A and so on.

So this is also a design time construct, also the join operation is fixed at design time. So, what changes over time is the position of the objects. For instance we may have object which is present in the security class A, and after a while this object becomes for instance less important or it becomes public domain and it can be moved to the security class C. To take an example from real life, what we see is that some top secret documents would be initially categorized as highly secure documents, but over a period of time this becomes public information and therefore, that particular document could then go to a lower security class.

So, you see that while the security classes are fixed as well as the flow of information is fixed among the security classes, the objects as such could move between security classes over a period of time.

(Refer Slide Time: 06:35)



So let us take some examples about information flow. Let us start with a very trivial case, which also is the most secure example for information flow, essentially because it does not allow information flow between classes. So, let us take this example (mentioned in above slide image) of a system where the security classes are defined by this particular set SC and it has 'n' security classes. Out of these the security class A_1 is the lowest while security class A_n is the highest, then we need to define the flow operator in this case it is A_i flows to A_i.

In other words what it means for every value from 1 to 'n', information can only flow within the class. In other words, it is not possible for information to flow from one class to another class. The third requirement is to define the join operator (mentioned in above slide image). So, in this particular case it is quiet trivial to know that if you combine a particular information from one class with an other information from the same class, will result in a new object which also belongs to the same class. Now let us look at a less stringent example, which is less secure than the previous case, which allows information only to flow from the low to high and not anywhere else.

So the security classes are defined as in the previous case i.e trivial case. So, we had 'n' security classes and A_1 is a lowest A_n is the highest and information can only flow from A_j to A_i where j is less than equal to i (mentioned in above slide in second case). So, what this means is that information can flow only from a lower class to a higher class, but the opposite direction from a high to low is not possible. Also while defining the join operator that is when we take information from a low class and combine it with information from a higher class, that is A_i with A_j we will get some new information which also belongs to the higher class that is the A_i class (3^{rd} point mentioned in above slide).

So, to repeat that if we take some information from a low class and combine it with an information in a higher class for instance A_2, then the new object that we create will also be in the A_2 security class. Thinks of what would happen, if instead of A_i over here (in 3^{rd} point) we had A_j. So, I leave that to you to think about.

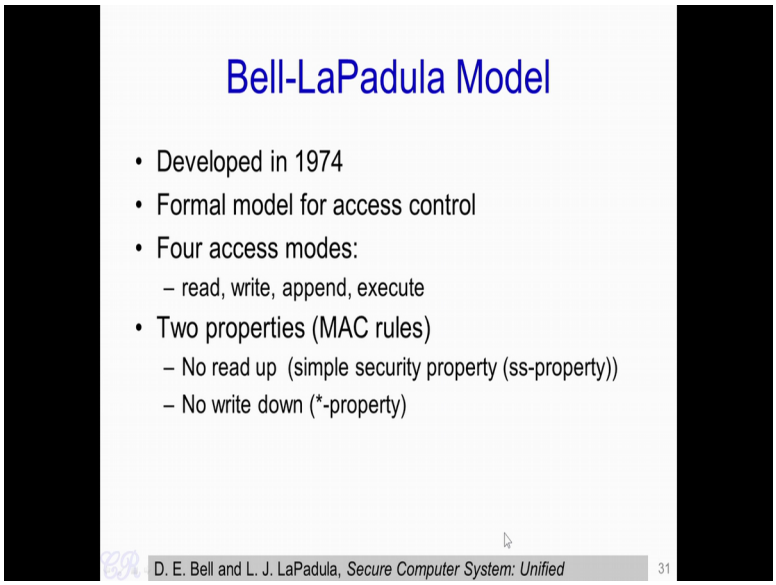
(Refer Slide Time: 09:34)

So now, let us come to the Mandatory Access Control Mechanism. So, essentially over here (mentioned in above slide image), the access mechanism or policies are based on regulations which are sent by a centralized authority. So, the most common form is the MLS or Multilevel Security policy. Essentially in this policy we have several access classes like the unclassified, confidential, secret and top secret. So, every object in the system needs a particular classification level. So, every object could be classified as either one of these 4 (i.e 4 classes mentioned above). Similarly every subject in the system also needs a clearance level. So, the clearance levels are also unclassified, confidential, secret and top secret, so one of these 4.

Now, a subject with clearance X can only access all objects in X and below X and not vice-versa, that is information can only flow upwards but cannot flow downwards. To take an example suppose we have a particular subject that is a user who has a clearance level secret, now what it means is that this particular user can access all the secret objects, all the objects which are classified as secret, all objects which are classified as confidential as well as unclassified, but this user will not be able to access any top secret objects.

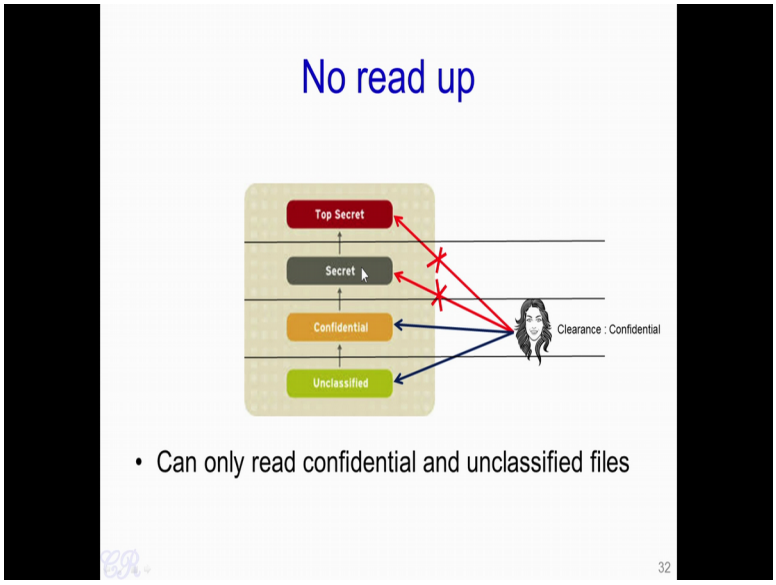
In other words information about from a top secret object cannot flow to a user with clearance secret, while the upward direction of information flow is possible. Now there are 2 types of MAC control techniques.

(Refer Slide Time: 11:36)



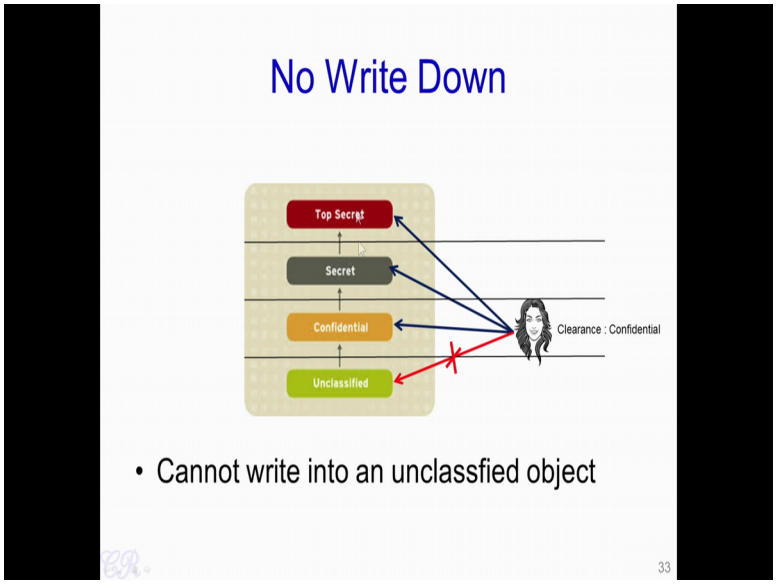
And the first we will see is the Bell LaPadula model. So, this was developed in 1974 and has is a formal model for access control. It gives four access modes read, write, append and execute. And it has 2 MAC properties that is No read up which is also known as the SS property or simple security property and No write down which is the star property (mentioned in above slide image). So, let us look at what these two properties are?

(Refer Slide Time: 12:09)



So, what this means that is the No read up that is the first property what we seen here (mentioned in above slide image), is that if we have a user with clearance confidential then that user can read all the objects which are confidential as well as all the objects which are unclassified. In the sense she cannot read any object which is classified as secret and any object which is classified as top secret. So, this particular mechanism does not allow read up.

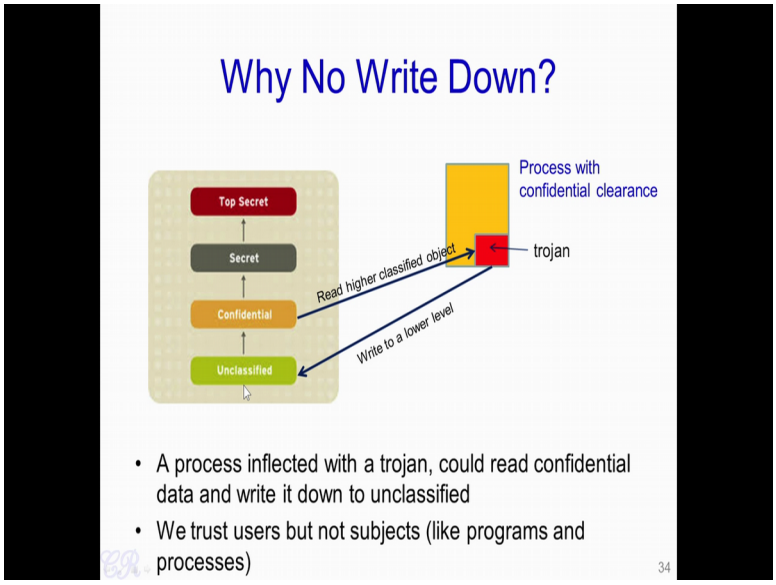
(Refer Slide Time: 12:46)



On the other hand the second policy's states that there is no write down. In other words this particular user with clearance confidential is allow to write or modify an object which is classified has confidential as well as this user can modify objects of classified as secret as well as top secret. She is able to write upwards, but what is not possible is that she is not allowed to write downwards (mentioned in above slide image).

So, she is not allowed to change or modify any object which is unclassified. Now this would seen very strange. So, let us see why such a mechanism was present.

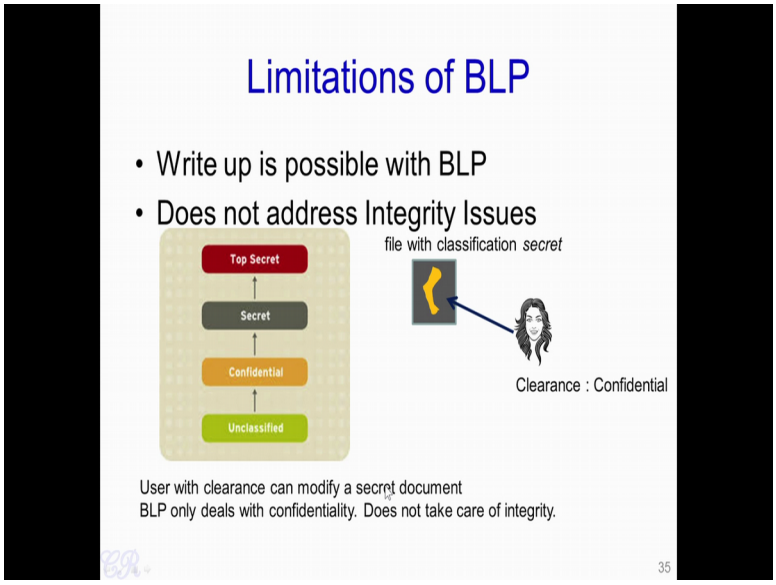
(Refer Slide Time: 13:35)



Essentially let us consider this particular scenario (mentioned in above slide image), where we have a process which is a confidential process with clearance confidential which is executing. Therefore, as we know it could read data or read an object which is also classified as confidential. Now let us assume that there is a trojan host that is present (mentioned above in red box). So, what we have seen is that when a trojan executes it's going to inherit all the clearance levels of the process.

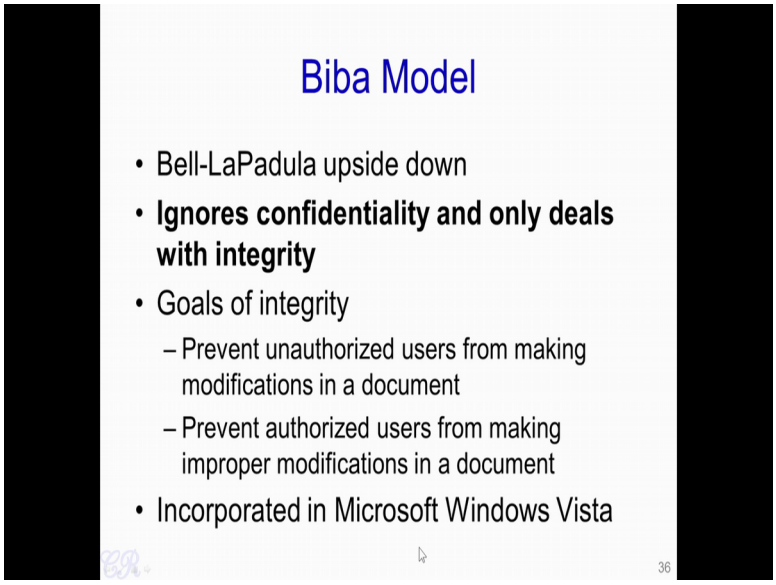
So in this case the trojan is also going to get a clearance confidential. Now assume that write down was allowed, in such a case it will not be very difficult for the trojan to read at particular file or a particular object which is marked confidential and write it down to the unclassified, that is it could be made as a public domain information. So, you see that this causes a flow of information from high a confidential level to a low confidential level, essentially through that particular Trojan (as mentioned above). Therefore, this particular model does not permit write down.

(Refer Slide Time: 14:58)



So, we look at the limitations of the Bell-LaPadula model, essentially what the Bell-LaPadula model does not prevent is that it allows a user with clearance confidential to write data upwards. So, this particular user with clearance confidential could essentially modify a particular top secret document or a secret document (mentioned in above slide image). In other words the BLP model does not address the integrity issues that are present. So, in order to cater to the integrity issues, there was another model which was used.

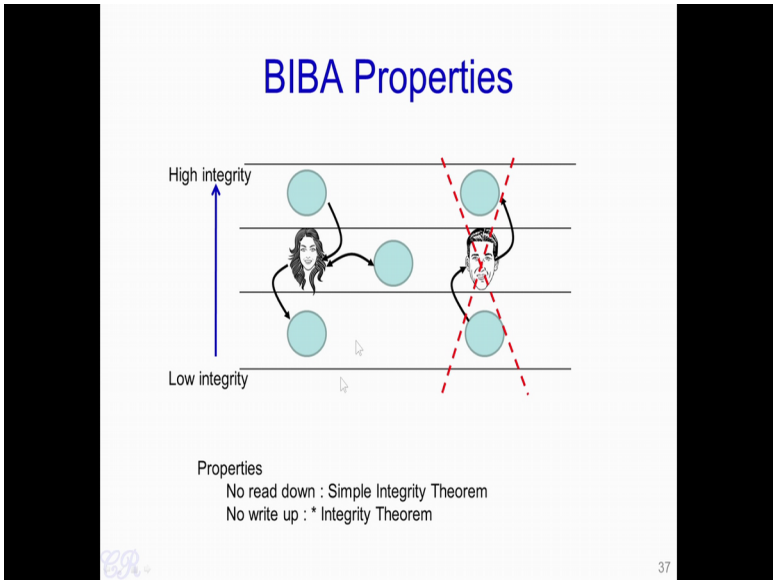
(Refer Slide Time: 15:46)



So, this is the Biba model (mentioned in above slide image). So, Biba model is the Bell-LaPadula model upside down. So while, the Bell-LaPadula model just focuses on confidentiality and ensures that no information flows from a high to a low.

The Biba model on the other hand ignores confidentiality all together and deals only with integrity. So, the main goal of the Biba model is to prevent unauthorized users from making modifications to a particular document. Also it prevents authorized users from making improper modifications in a document. So, this Biba model is incorporated in Microsoft windows vista operating system.

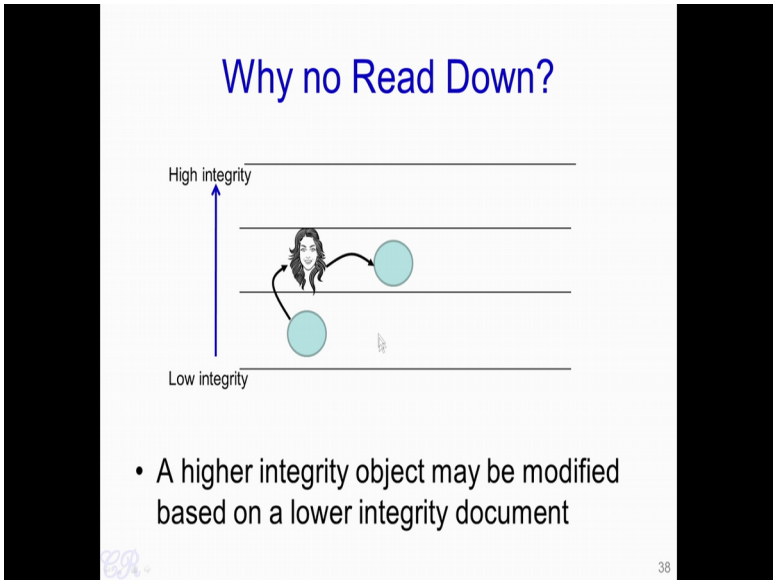
(Refer Slide Time: 16:38)



So, what the Biba model defines is that the user can first of all read and write to any object within the same security class, and the user could write to a object in a lower security class, and read from an object present in a higher security class. This particular object (mentioned object in top class) can be read, this particular object (object in lower class) can be modified because it's in a lower security class, this can be read because it's in a higher security class, while objects in the same security class can be read and written. Please follow the arrows in this particular case (mentioned in above slide image). So, what it does not allow is that the lower security class be read from. Thus essentially a user cannot read from the lower security class and cannot write to a higher security class.

So, you see this is exactly the opposite of what the Bell-LaPadula model tells us. So, the properties of the Biba model, is that there is no read down and no write up this is the star integrity theorem and while the no read down is a simple integrity theorem (mentioned in above slide image). So, with respect to the security classes this is the low integrity (low level class) and this is the high integrity class (high level class).

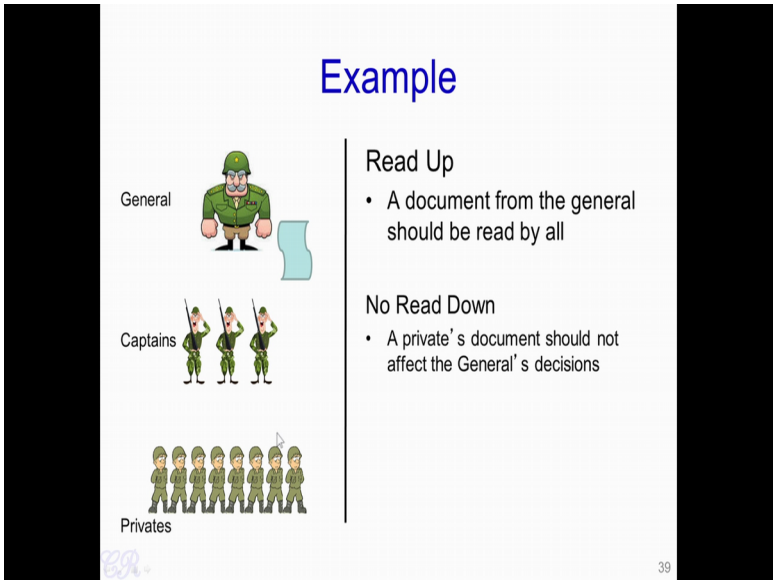
(Refer Slide Time: 18:05)



So, why does the Biba model not support read down? Why cannot a user with a particular security class read from an object from a lower security class? So, why is this (low to high level) particular direction for information flow not permitted? The reasoning behind that is that a higher integrity object such as this one (object present in middle in above slide) may be modified based on a lower integrity document.

So for instance since, this particular user (mentioned in above slide) with a particular security class is capable of writing to this particular object present in the same security class. Now if she is able to read from a lower security class, and there is a flow of information upwards what it means is that she can then get influenced by this information (information present in lower security class object) or she can then copy this information on to this particular object in the higher security class. So, this means a lower integrity object is affecting a higher integrity object, and therefore the Biba model prevents such flow of information.

(Refer Slide Time: 19:24)



So to take an example, let us say the hierarchy in the military where you have a general right on top, then the captains and the privates who are right at the bottom of the hierarchy (mentioned in above slide).

Now, the Biba model allows read up meaning a document which is prepared by the general should be read by all, that is a document which is created by the general should be read by the captain as well as the privates. However, no read down is permitted, that is a document written or modified by the privates at the lower end of the hierarchy should not affect the general's decision.

(Refer Slide Time: 20:08)

So, we see that in spite of having such flow mechanisms, where information can be restricted in how they flow that is we can prevent information flow between security classes. However, due to various reasons and due to bugs or other flaws in the design, it may be possible for information to actually flow between security classes, in spite of having such robust measures like the access control measures that we just discussed.

So, what we will do in the next video is that, we will look at such techniques where information can actually flow in spite of having such mechanisms in place. So, we will look at control flow hacking, essentially buffer overflows and how they could be used to allow an unauthorized user gaining information from a system.

Thank you.

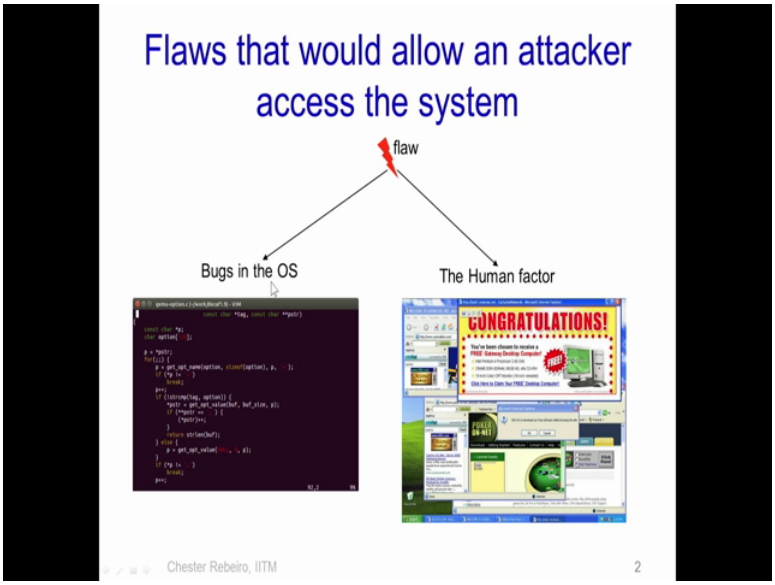
Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 08
Lecture – 37
Operating System Security (Buffer Overflows)

Hello. In this video, we will talk about Buffer Overflows. Essentially, Buffer Overflows is a vulnerability in the system and it is not just restricted to the operating system, but it could be pertaining to any application that runs in the system. Now buffer overflows is vulnerability that allows malicious applications to enter into the systems, even though they do not have a valid access. Essentially it would allow unauthorized access into the system.

So let us look at Buffer Overflows in this particular lecture.

(Refer Slide Time: 01:02)



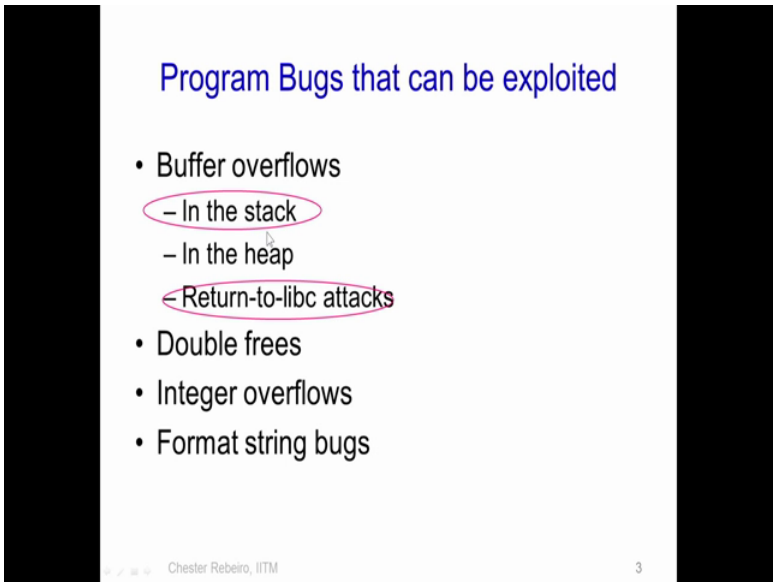
So, when we look at how an unauthorized user or an unauthorized attacker could gain access into the system so, we see that it is just by flaws present. There are two types of flaws that a system can have (mentioned in above slide image). One it could have Bugs present in the application or the operating system in this particular case or it could have

flaws due to the Human factor. When we for instance browse the internet where we see many such web pages opening and prompting us to click on particular things which would take us to may be a malicious website and a result of that would cause malicious applications to be downloaded into the system.

Another one, which is more pertaining to the operating system, is when there are bugs in the operating system code. Now, modern day operating systems especially the ones that we typically use on a desktop and servers are extremely large pieces of code. For instance, the current Linux kernel has over ten million lines of code and all these codes are obviously written by programmers and will have numerous bugs. So, these bugs are not very easy to detect; however, if an attacker decides to look, he could find such a bug and he could then exploit this bug in the operating system to gain access into the OS.

And as you know that once the attacker gains access into the OS, he will be able to do various things like he will be able to execute various components of the operating system code, he could control all the resources present in the system, he could also control which users execute in the system and so on. Thus, an unauthorized access through a bug in the operating system is a very critical aspect.

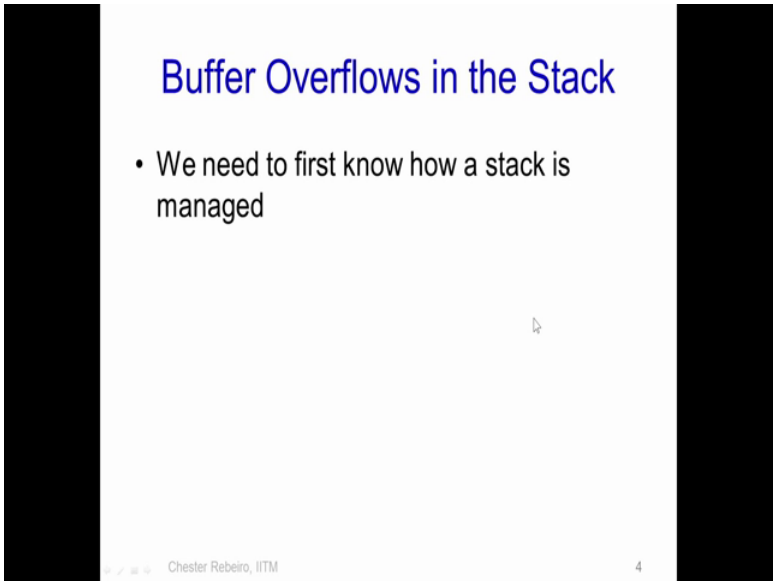
(Refer Slide Time: 03:20)



So there are a number of bugs that an attacker can exploit in order to gain unauthorized access into the operating system. So, here is the list of some of them (mentioned in above slide image). So, there could be buffer overflows in the stack of the program or in the OS, in the heap, there may be something known as Return-to-libc attacks. There are double frees, essentially this occurs when a single memory location which is dynamically allocated through something like a malloc gets freed more than once, there are integer overflow bugs, and there are format string bugs.

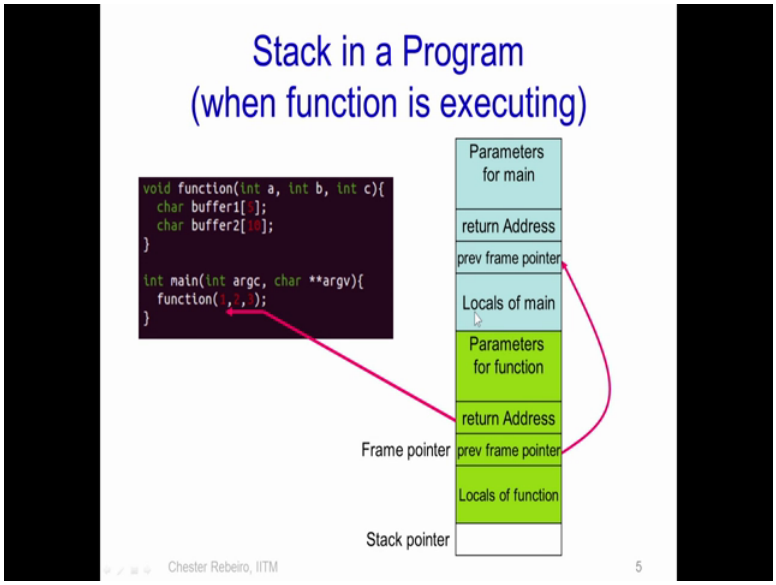
So there are essentially numerous different ways that bugs can be exploited by an attacker to enter into the system. So, what we will be seeing today are the bugs in the stack and something known as a Return-to-libc attack which essentially is a variant of the buffer overflow attack in the stack.

(Refer Slide Time: 04:22)



So, in order to understand how the buffer overflows work in the stack, we first need to know how a stack is managed. So let us see how the user's stack of a process's is managed.

(Refer Slide Time: 04:38)



So let us say we take this very simple example (as mentioned in above slide) which has two functions the main() and in this main function, we invoke another function with parameters 1, 2 and 3 (i.e function(1, 2, 3)). And this function just allocates two buffers - buffer 1[5] bytes and buffer 2[10] bytes. So, as we know when we execute this program in the system, the operating system creates a process comprising of various things like the instruction area containing the text or the various instructions of this particular program, the data section, the heap as well as the stack.

The stack in particular is used for passing parameters from one function to another and it is also used to store local variables. So let us see how the stack is used in this particular example. So let us say that this is the stack (mentioned in above slide image) and this stack corresponds to when the main function is executing. Now when main() wants to invoke this function over here that is function(1, 2, 3); it begins to push something on to the stack. So, what is pushed onto the stack we will see now?

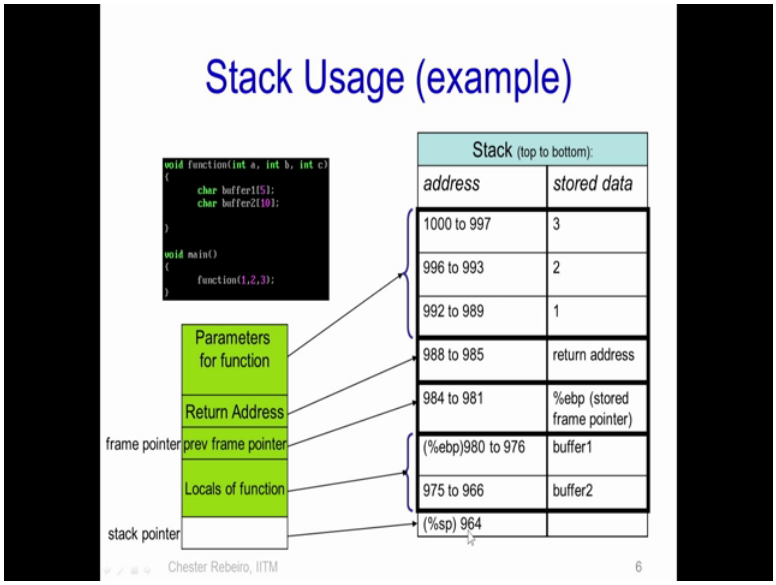
So, first the 3 parameters 1, 2 and 3 which are passed from main to function (mentioned in program) are pushed on to the stack that is parameters for function 1, 2 and 3 would be pushed onto the stack. Then the return address is pushed on to the stack. So, this return address (as mentioned in stack) will point to the instruction that follows this function

invocation (i.e calling function mentioned in main()). As we know in order to invoke function in an x86 space processor, the instruction that is used is the call. So the return address will point to the next instruction following the call.

So, after the return address (mentioned in stack) something known as the previous frame pointer is push onto the stack, this frame pointer points to the frame corresponding to the main() function. So, this is the frame which is used to when the function is executed (green part of stack as mentioned in above slide image), while this frame (blue part of stack) is used when main is executed. Now after the previous frame pointer is used, the local variables which are defined in function are then allocated. In this case, we have two character arrays which are allocated 1 is of size 5 and the other is of size 10 bytes i.e buffer 1[5] and buffer 2[10].

So besides all of this, we have 2 CPU registers which are used to manage this stack pointer, one is the frame pointer which is typically the register bp in Intel x86, and the other one is the stack pointer or sp in the x86 nomenclature. Now the frame pointer points to the current functions framed (green part frame). So, it actually points to this particular thing (green part frame of stack) corresponding to the frame for function(). Now after this function (mentioned in program) completes its execution and returns, this previous frame pointer is loaded into the register bp, therefore the frame pointer will then point over here that is the frame corresponding to the function main (blue part frame). Now the stack pointer on the other hand points to the bottom of the stack.

(Refer Slide Time: 08:37)

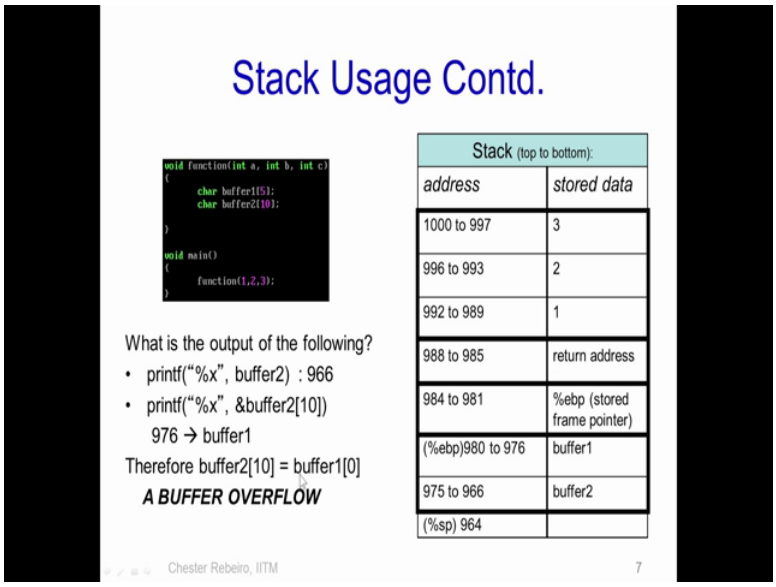


Now, let us look at this in more detailed. Let us say that this is the stack (mentioned in above slide image) and this is the address for the various stack locations (first column in stack) and this is the data stored in that particular address (second column in stack). So let us assume that the top of the stack is 1000 i.e address and it decrements downwards. So, this was the stack (stack mentioned in above image in green) corresponding to the function when it is invoked. So, we first see that there are the parameters that are passed to the function are pushed onto the stack; this is the parameters 3, 2 and 1 (mentioned in stored data in stack above) which are pushed onto the stack. So, we note that each of these parameters since they have defined as integer in this function are given 4 bytes. So the integer 'a', which is passed to function would start at the address location 997 and from there be 4 bytes 997, 998, 999 and 1000. Similarly, the second and third parameters also take 4 bytes (refer in above image).

The return address for this function (function in program) at essentially the point at which the function has to return is also given 4 bytes (i.e address 988 to 985). While the base pointer since it is a 32 bit system is also given 4 bytes (i.e address 984 to 981), then we have the buffer1[5] which is allocated as a local of the function, which is given 5 bytes 976 to 980 and then buffer2[10] is allocated 10 bytes. So, these two arrays are the locals of the function. The base pointer points to this particular location (i.e address 984

to 981) and the stack pointer points over here (mentioned in slide image) to the address number 964.

(Refer Slide Time: 10:42)



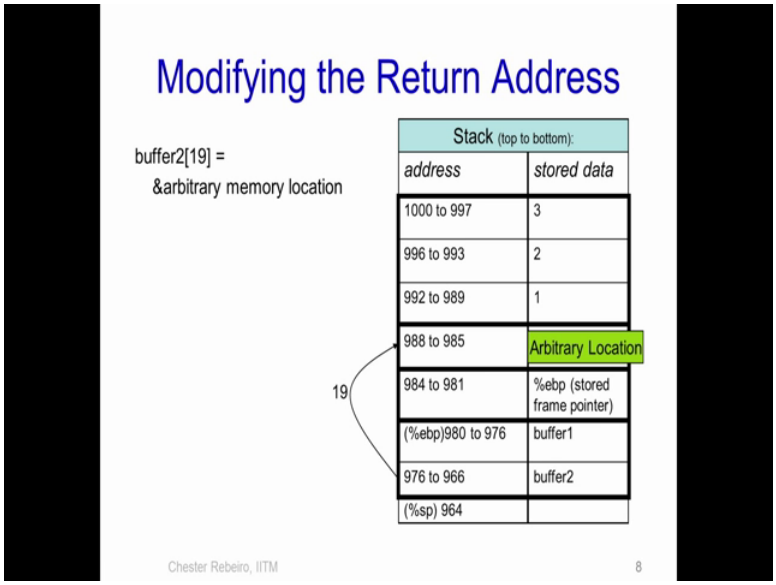
Now, let us look at some very simple aspects. So let us say what would happen if we print this particular line, so printf("%x", buffer2). So, as we know buffer 2 corresponds to the address of this particular array, so this particular printf statement would print the address of buffer 2. So, if we look up the stack we see that the start address of buffer 2 is 966; therefore, this printf will print 966.

Now what happens if we do something like this, printf("%x", &buffer2[10])? So, we know that buffer 2 is of 10 bytes and will have indexes from 0 to 9; now buffer 2[10] is 966 (i.e address) plus 10 which is 976. So, what is going to be printed over here is 976 (i.e address of buffer2[10]). Now it so happens that 976 is outside the region of buffer 2, in fact 976 is in buffer 1 (refer slide time 10:42). Therefore, what we are getting now is that we are printing an address which is outside buffer 2, and this is what is known as a Buffer Overflow.

Essentially, we have defined a buffer of 10 bytes, but we are accessing data which is outside the buffer 2 area. So, we are accessing the 10th, 11th, 12th and so on byte. So,

this is known as a buffer overflow. Now, what we will see next is how this buffer overflow can be exploited by an attacker, and how an attacker could then force a system to execute his own code.

(Refer Slide Time: 12:39)

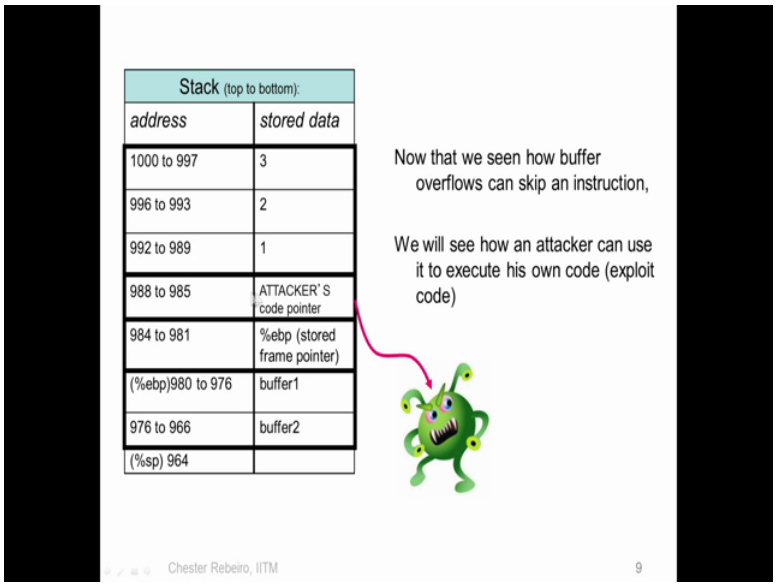


Now one important thing from the attackers' perspective is the return address. If the attacker could somehow fill this buffer 2 in such a way that he would cause a buffer overflow, and modify this particular return address (address mentioned in stack), then let us see what would happen. So let us say he makes this particular statement. So, buffer2[19] is some arbitrary memory location. So, what the attacker is doing is that he is forcing this buffer 2 to overflow and he is overflowing it in such a way that the return address which was stored onto the stack is replaced with his own filled location (i.e address 988 to 985 is replaced with arbitrary location).

After the function completes executing, it would look into this location and instead of getting the valid return address it would get this arbitrary location, and then it would go to this arbitrary location and start to execute code. So, what we would see is that instead of returning to the main function as would be expected in the normal program, since the attacker has changed this return address to some arbitrary location, the processor would then cause this instructions corresponding to this arbitrary location to be fetched and

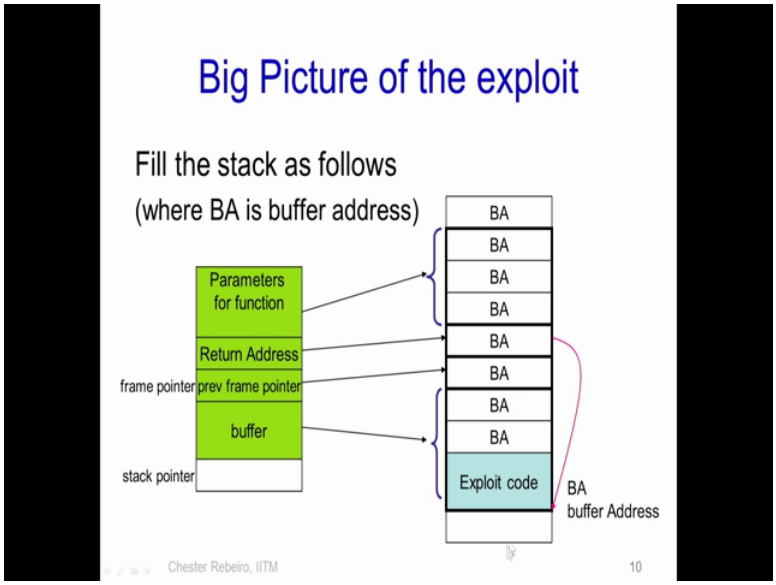
execute it. So, now, it looks quite obvious what the attacker could do in order to create an attack.

(Refer Slide Time: 14:19)



So essentially what the attacker is going to do is that he is going to change this return address the valid return address present in the stack with a pointer to an attack code (i.e attacker's code pointer). Therefore, when the function returns, instead of taking the standard return address it would pick the attackers code pointer (as mentioned in stack in place of valid return address); and as a result, it would then cause the attackers code to begin to execute. So now, we will see how the attacker could change the return address with it is own code pointer.

(Refer Slide Time: 14:56)

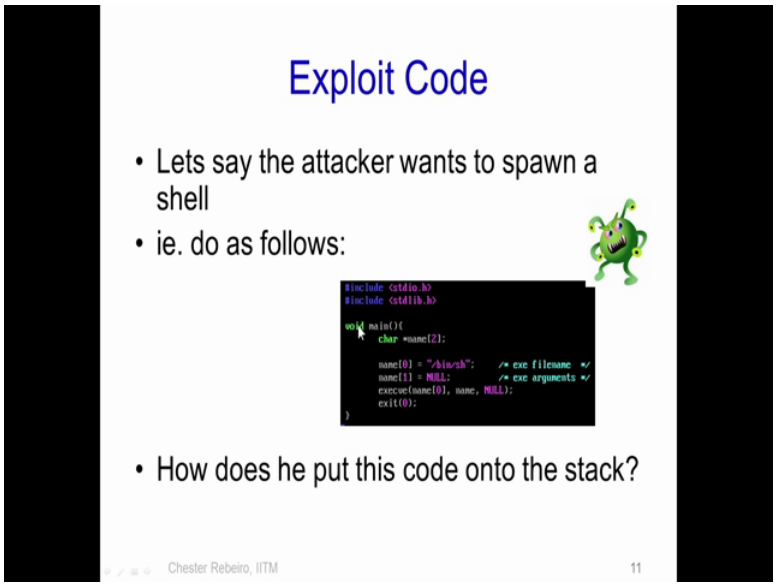


In order to do this, what we assume is that the attacker has access to this particular buffer (buffer mentioned above in green stack). So, this means that the attacker will be able to fill this particular buffer as required. For instance, this buffer could be say passed through a system call, for instance, if the operating system would require that a character string be passed through the system call by the user, and thus the attacker will be able to fill that character string and pass it to the kernel. So, what the attacker is going to do is to create a very specific string that it has the exploit code, and it is also able to force the operating system or for that matter any application to execute this exploit code. So, how it is going to do it is as follows.

So, in the buffer, the attacker will do two things; first he will put the exploit code, that is the code which the attacker wants to execute in the lower most region of the buffer, and then begin to overflow the buffer (mentioned in above slide image). Essentially what he is going to put is this address location BA. Now BA here is the address of this exploit code. So, it is assumed or if a smart attacker will be able to determine what the address location is of buffer, and he will overflow the buffer with BA. So, he keeps overflowing the buffer with BA, and when this happens at one particular case the return address present in the stack is changed from the valid return address to B A.

Thus, when the function returns what the CPU is going to see is the address BA in the return address location. Thus it is going to take this address BA and start executing code from that. So, since BA corresponds to the address where the exploit code is present, the CPU would then begin to execute this exploit code and in this way, the attacker could force the CPU or the processor to execute the exploit code.

(Refer Slide Time: 17:33)



So, now we will see how a one particular attack code is created, and how an attacker can force an application or an operating system to execute that exploit code. So let us take a very simple example of the exploit code, which is shown over here (mentioned in above slide). Essentially this exploit code which we call as shell code does nothing but only executes a particular shell, the shell is specified by /bin/sh.

And this particular function execve is invoked in order to execute this shell. The parameters are name[0] which comprises of the executable name and there is name[1] which is essentially a null terminated string. So, we will see how this particular code can be forced to be executed by an unauthorized attacker. So the first question that needs to be asked is how does this attacker manage to put this code onto the stack?

(Refer Slide Time: 18:48)

The first step in doing so is that the attacker needs to obtain the binary data corresponding to this particular program (first program in above slide image). In order to do this, what the attacker does is that he will re-write this program in assembly code. So, this assembly code as we see here (assembly code mentioned in above image) does exactly the same thing as done by this program (first basic program).

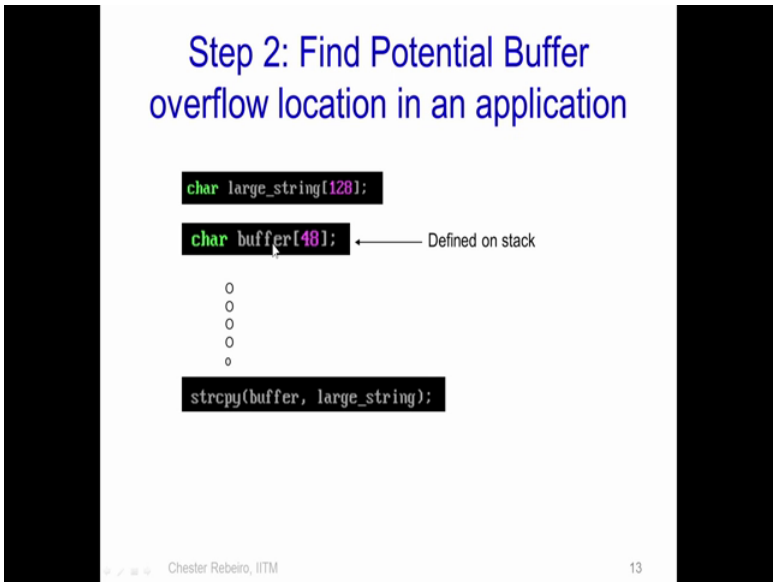
The next thing what the attacker would do is to compile this particular assembly code and get what is known as the object dump (file pointed by assembly code in above image). So the object dump is obtained by running this particular command thing (first command mentioned above). So, he first compiles this particular code to get what is known as the shellcode.o which is the object file, and then he will run this particular command which is objdump-disassemble-all shellcode.o to get this particular file (file pointed by assembly code in above image).

Now, what is important for us over here is this particular column or the second column (mentioned in above file). So the numbers what you see over here, the hexadecimal numbers are in fact the machine code for this program (first program). So the numbers like eb 1e, 5e, 89 76 08, and so on correspond to the machine code of this particular program. In other words, if the attacker manages to put this machine code onto the stack

and is able to force execution to this particular machine code, then the attacker would be able to execute the shell as required.

So the machine code is shown over here (second point mentioned in slide time 18:48) and one thing which is required for this particular attack is to replace all the 0's present in this machine code with some other instructions, so that you do not have any 0's present over here.

(Refer Slide Time: 20:48)

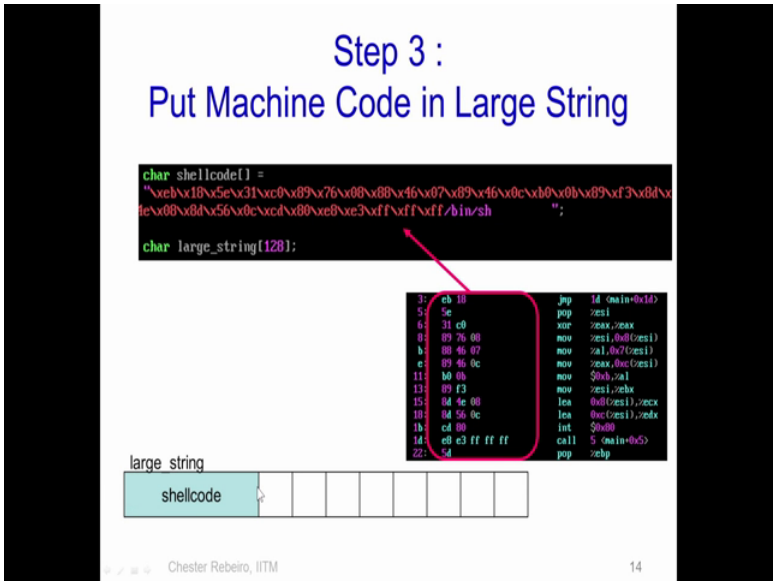


The next thing is to scan the entire application in order to find one location which can be exploited for a buffer overflow. So, essentially the requirements for a buffer overflow is that the attacker finds in the application code a command such as this a strcpy(buffer, large_string) where buffer is a small array and it is defined locally in the stack, while the large string is a much larger array (mentioned in above slide image). So, as we know the way this particular function strcpy() works is that the large string gets copied to buffer and this copying will continue byte by byte until there is a /0 which is found in large string; in which case the strcpy will complete executing and will return.

So let us assume that the attacker has found such a case where we have the buffer, a large string and a string copy, and the buffer is a small array defined onto the stack and how

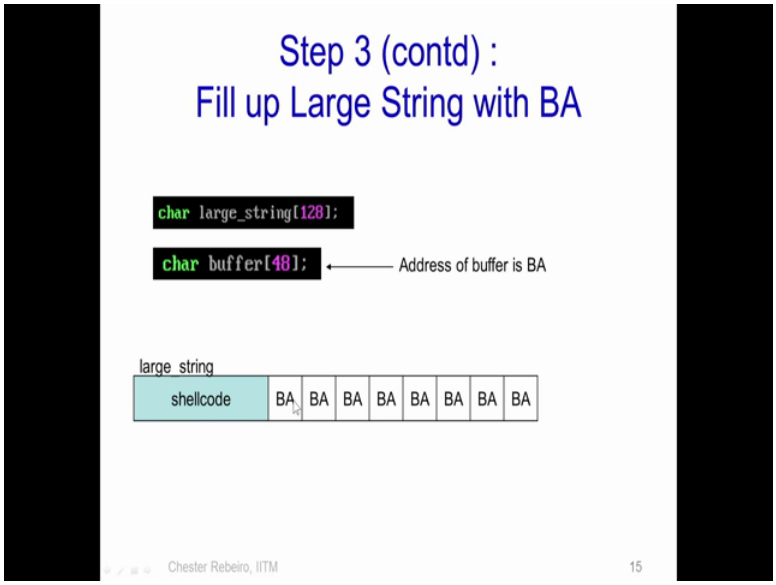
does the attacker make use of this.

(Refer Slide Time: 22:07)



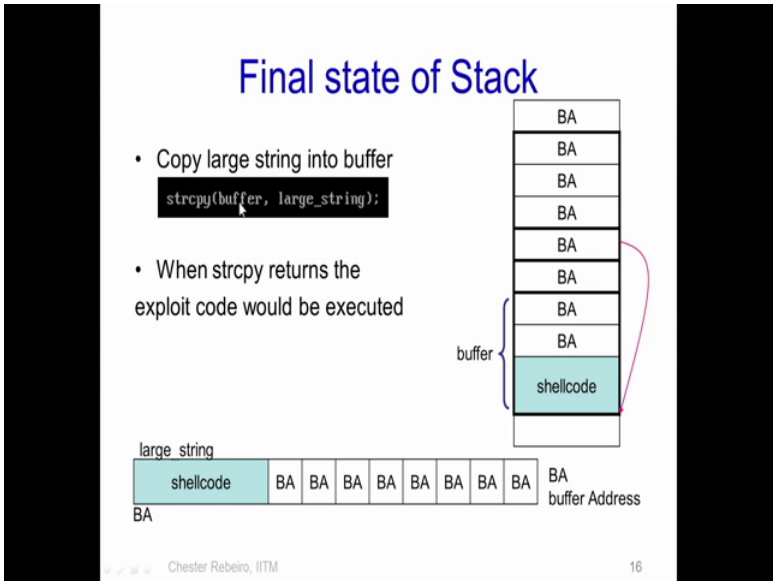
So, what the attacker would then do is create something known as the shell code array which essentially is the code that he wants to execute. So, he creates the shell code array comprising of all the assembly op codes or machine codes which it wants to execute and he places this code or this shell code in the first part of the large string. So, if you look at this the large string array, which is a very large string of 128 bytes in this case, gets the shell code in the first part (mentioned in above slide image).

(Refer Slide Time: 22:46)



Then he computes what the address of the buffer should be, and fills the remaining part of the large string with the buffer address (as mentioned in above slide image).

(Refer Slide Time: 23:00)

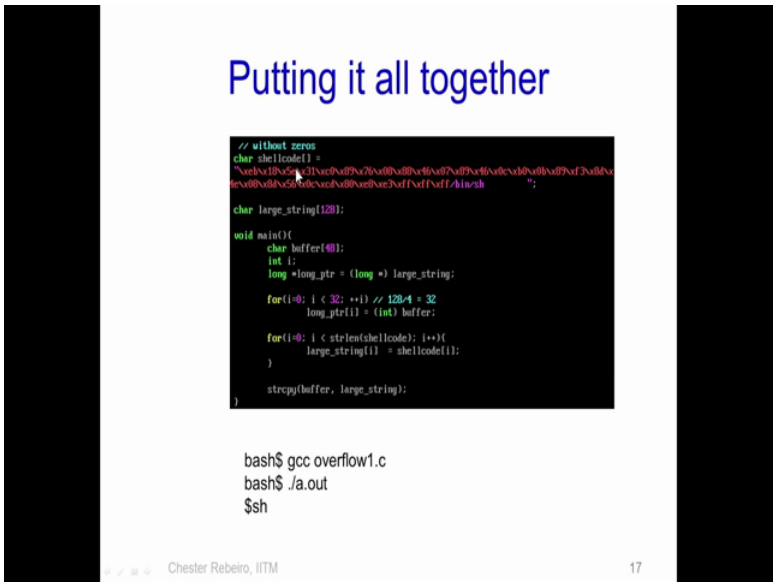


Now he needs to force the string copy to execute with this buffer and the large string which he has just created as shown over here (i.e strcpy(buffer, large_string)). Now as a

result of this string copy being executed, there is a stack frame created for the strcpy() and as we know that the strcpy() will continue to copy bytes from the large string to the buffer until it finds the /0. So, in such a way what would happen onto the stack is that the large string gets copied.

So first, the shell code gets copied and then the buffer address keeps getting overflowed on to the buffer and keeps going on until a /0 is found (stack shown in above slide image). So, when the strcpy() executes what we have seen is that instead of getting the valid return address, it now gets what is known as the buffer address. So, essentially we know that the buffer address points to this particular location (i.e shellcode location), and therefore, the CPU would be forced to return to this location pointed to by the buffer address and execute the shell code. As a result the attacker would be able to execute the shell code which in this particular example was the exploit.

(Refer Slide Time: 24:22)



So let us look at the entire thing all together. So we have the shell code which is the code which the attacker wants to execute (shellcode mentioned in above slide image). Over here (in above slide), we have just defining it as a global array, but in reality this could be entered through various things like a scanf() or it could also come in through the network card, essentially we passed a packet with particular format containing the

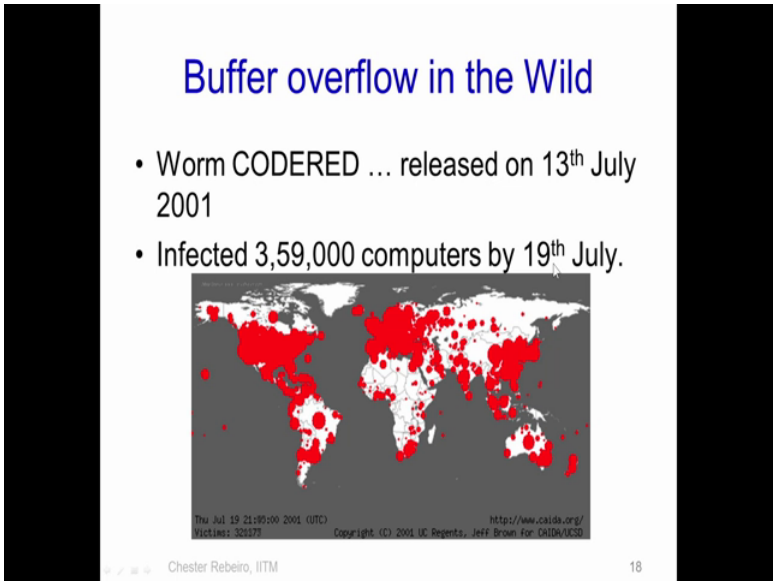
exploit and various other different ways of passing in the shell code.

Next, let us assume that somewhere in the application there is this particular code (code mentioned in above slide inside main()). We have the large string and we have short a string which is buffer, which is a local array locally defined array and therefore, gets created onto the stack.

So, what we first do is somehow manage to fill the long string or the large string with the address of buffer. So, if you recollect, this is the BA parts which are present (first for loop instruction), then we will copy the shell code on to the large string (second for loop instruction). So, we have created this large string in the format that we require. In the first part of this large string is the shell code, and then it is followed by the buffer return address. And then if there is a function like strcpy() which copies large string into buffer it will result in a buffer overflow to occur and instead of the function returning to this particular point soon after string copy.

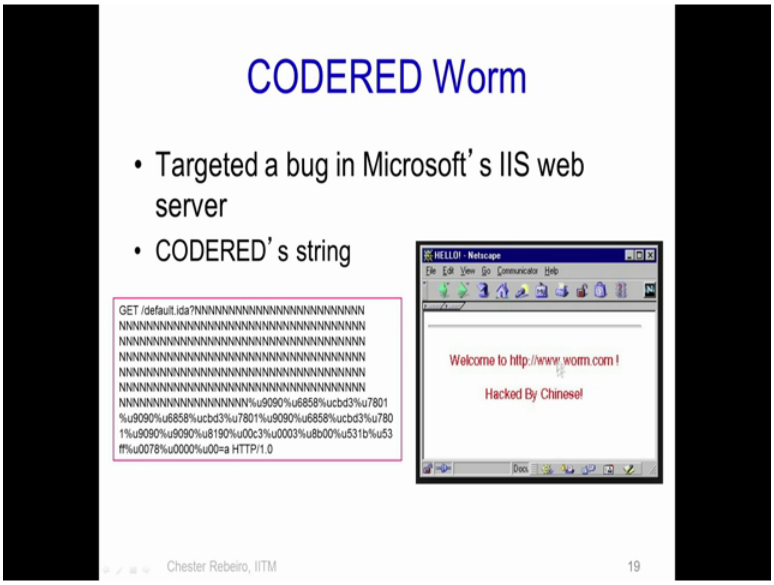
On the other hand, execute this particular shell code and cause this shell specified by this command to be executed i.e /bin/sh. So, if we actually see this, if we execute gcc overflow1.c and run ./a.out instead of just doing this strcpy and exiting, this particular program created a new shell due to the exploit code that is executing.

(Refer Slide Time: 26:44)



So, buffer overflows are an extremely well known bug and extremely exploited by various different malware and viruses over the last decade or actually more than a decade. And one of the first viruses that actually use the buffer overflow was the worm called CODERED which was released on July 13th 2001. So, this created a massive chaos all over the world, and the red spots actually show how the virus spread across the world in about or rather in less than a day or a few hours. So, we see that this particular virus which used the buffer overflow infected roughly 359000 computers by July 19th 2001.

(Refer Slide Time: 27:36)



So essentially the targeted application by this worm or this particular worm was the Microsoft's IIS web server and the string which was executed was as shown over here (mentioned in above slide image). So, this string was actually the exploit code which was executed and what it resulted was something like this being displayed in the web browser.

Thank you.

Introduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week – 08
Lecture – 38
Preventing Buffer Overflow Attacks

Hello. In the previous video, we had seen how attacker essentially an unauthorized user of the system could cause his own code to execute in the system by exploiting what was known as a buffer overflow bug. So we had seen, at least we had created such a buffer overflow bug and shown how the attacker could create his exploit code and run that exploit code which created a shell in the system.

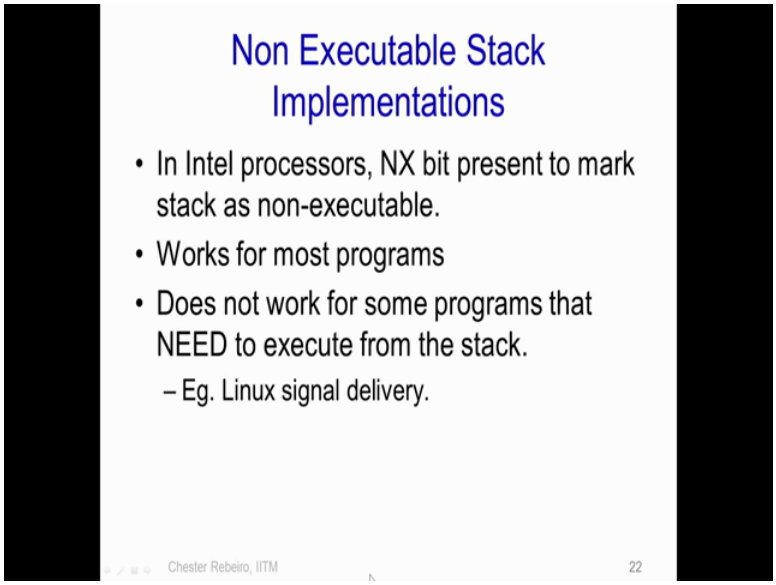
So, this particular example (mentioned in below slide image) was an application based example in the user space, but very similar kind of exploits can be written in the operating system. So, what we will be seeing in this video are techniques of how this buffer overflow vulnerability is overcome, and also how the attack has progressed over the years and evolved over the years, to make more powerful attacks in order to overcome these protections.

(Refer Slide Time: 01:32)

The first and the most obvious way to prevent the buffer overflow attack which occurs in the stack is by making the stack pages non-executable. What we seen is that the attacker would force the CPU to execute an exploit code which is also present in the stack. So, for instance over here (inside main() in above image), this buffer which is defined on the stack also contain the exploit code and the string copy was executed in such a way that after string copy completed its execution, it would cause the exploit code that is the shell code which is present on the stack to be executed.

So, one obvious way to prevent this attack is to make the stack pages non-executable, so and this is what is done in systems that are used these days. So, if you would actually run this particular program on your Intel systems, and instead of getting the exploit code to execute, you would get a segmentation fault. This would be caused because the program is trying to execute some instructions onto the stack.

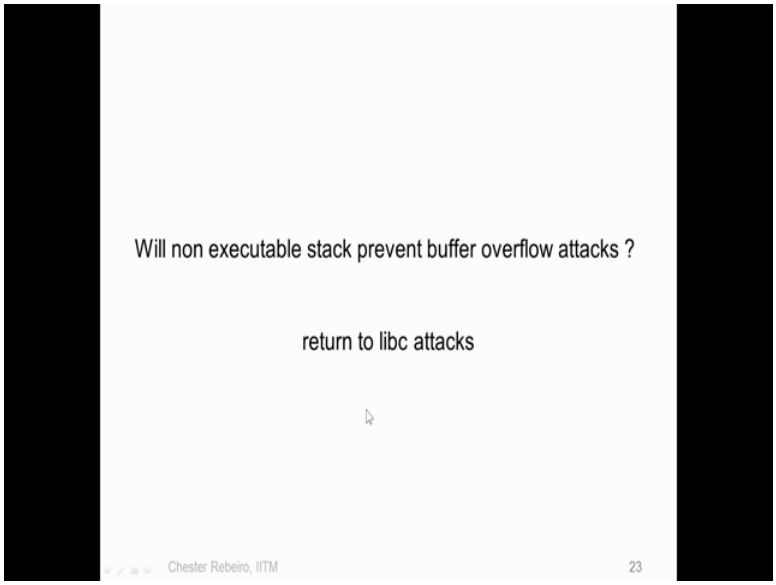
(Refer Slide Time: 02:58)



In Intel machines, an NX bit is present in the page tables to mark the stack as non-executable. While this works for most of the programs that is in most programs it is an added benefit, but the problem is some programs even though they may not be malicious require to execute from the stack. So these programs need to execute from the stack in order to function properly. Therefore, setting the NX bit is not always very beneficial for

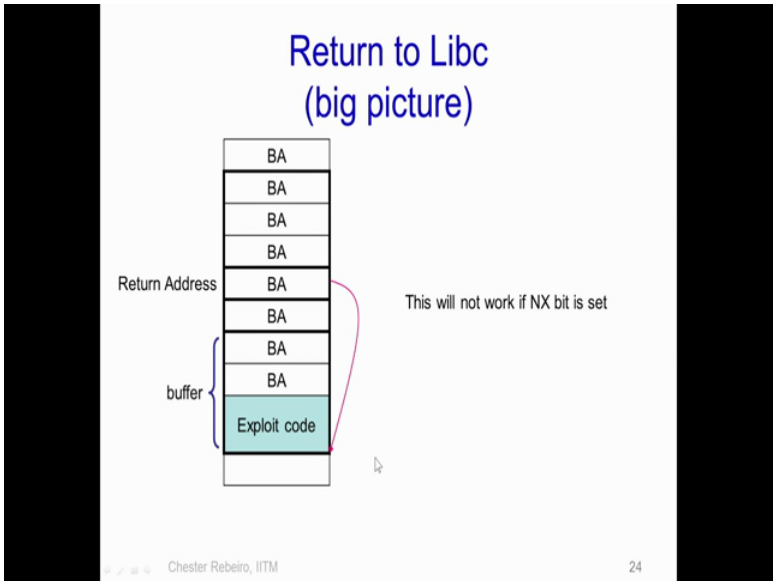
all programs.

(Refer Slide Time: 03:37)



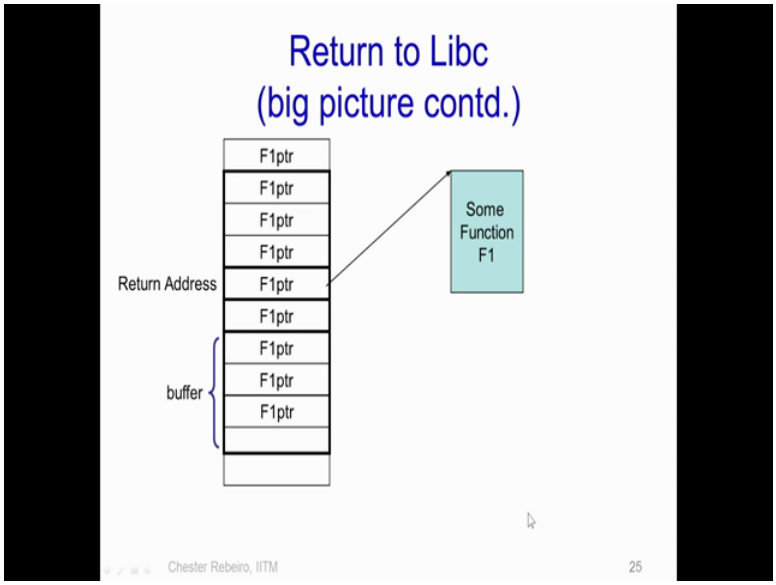
So the next thing, we are going to ask ourselves is that if we make the stack as non-executable, will it completely prevent buffer overflow attacks and in fact, it does not. Over the years buffer overflow attacks have evolved to something known as the return to libc attacks, which could be used even on systems which have a non-executable stack.

(Refer Slide Time: 04:13)



So let us see in very brief how a return to libc attack works (refer above slide image). So, essentially what we have done, when we try to overflow the buffer in the stack is that we had the exploit code present onto the stack, and we had replaced the valid return address with the address of the buffer. Now this did not work for us in modern day systems, because the stack was set to non-executable. So let us look at how return to libc works in spite of having the non-executable stack that is in spite of having the NX bit set.

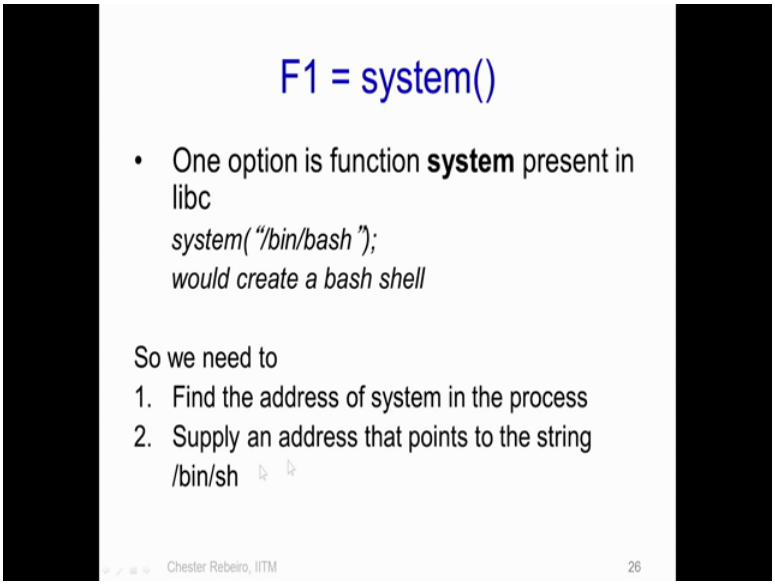
(Refer Slide Time: 04:56)



So, what the return to libc does is that instead of forcing the return address to branch to a location within the stack, it forces the return address to go to some other location containing some other function. Let us say this function is F 1 (mentioned in above slide image). So, what is filled onto the stack through a buffer overflow is a pointer to F 1 i.e F1ptr therefore when the function completes executing, the return address taken from here is a F1ptr, and therefore, the CPU is forced to execute this function F 1.

Now, the next question is - what is this function F 1 (blue box in above slide)? So, one thing is certain that this function cannot be the attacker's own exploit code. So, it has to be some valid function which is already present in the code segment and which has the permission to execute. So, what would this function F 1 be? That is point number 1; and point number 2 is how will an attacker use a normal function which is present in the program to do something malicious such as to run and exploit code?

(Refer Slide Time: 06:23)



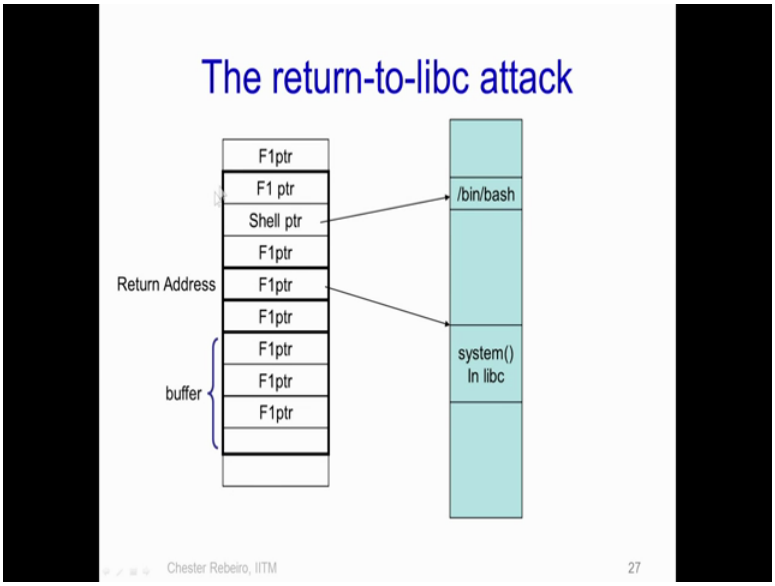
There are various ways in which the function F1 can be implemented, but what we will see today is the function F 1 implemented using this particular function called system(). Now, system is a function which is present in the library called libc, and what it does is that it takes a character pointer to a string which would take an executable. So the string which is passed to system takes an executable name (i.e /bin/bash as mentioned in above slide) and when executed it would essentially execute this particular program(i.e bash).

So, in this particular example (i.e system("/bin/bash"), what system would do is that it would execute the bash shell, thereby creating a bash shell. Now libc is a library, which is used by most programs. So, even a normal "hello world" program that you write would quite likely use the libc library. So, what this means is that in your process's address space, there is the function system(). So, what the attacker would need to do now is to just identify in your process's address space where the system() function resides. Next he needs to somehow pass an argument to this (system function) in order to run a program.

So, suppose let us say if we are continuing the example, what we seen in the previous video where the attacker creates an exploit which executes a shell. So, in this case also, if the attacker wants to execute the same shell, then in addition to finding the systems

functions address in the memory space, the attacker would somehow needs to pass this particular string /bin/sh as a parameter to the system. So, this means that the attacker would need to find an address that points to the string /bin/sh.

(Refer Slide Time: 08:32)

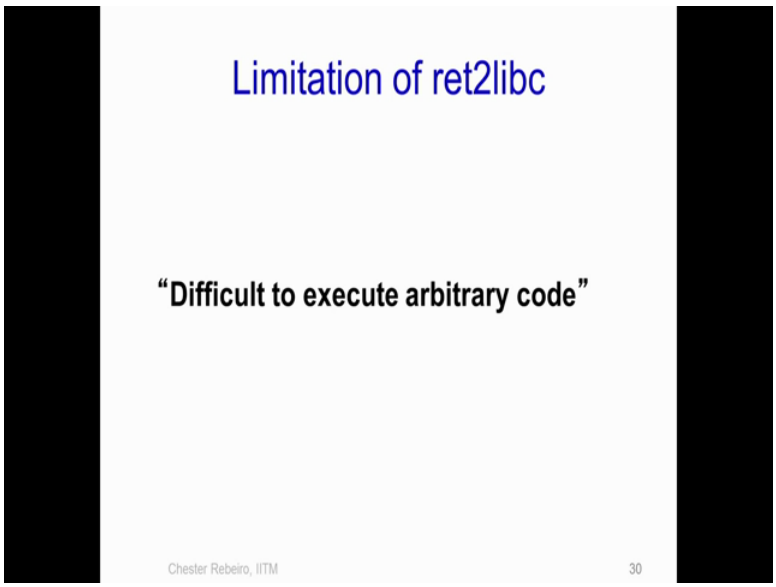


So, what is typically done is have a stack frame something like this (mentioned in above slide image) where the return address or rather the valid return address is replaced by F1ptr which is a pointer to the system function which is present in libc and if your program uses the dynamically linked library libc, then such an address will be valid. Second, what the attacker needs to do is to pass an argument to the system function, which essentially has an executable name.

So, essentially somewhere in your address space (blue box mentioned in above image), the attacker needs to find this particular string being present that is /bin/bash or /bin/sh, and fill in the stack with the pointer (i.e Shell ptr) to this particular string. So, essentially, it requires two things; one is the return address in the stack to be modified with the address of the system call which is present in libc, and also the parameter passed to libc should be present on the stack. So, in this case it is a Shell ptr which points to this particular string /bin/bash.

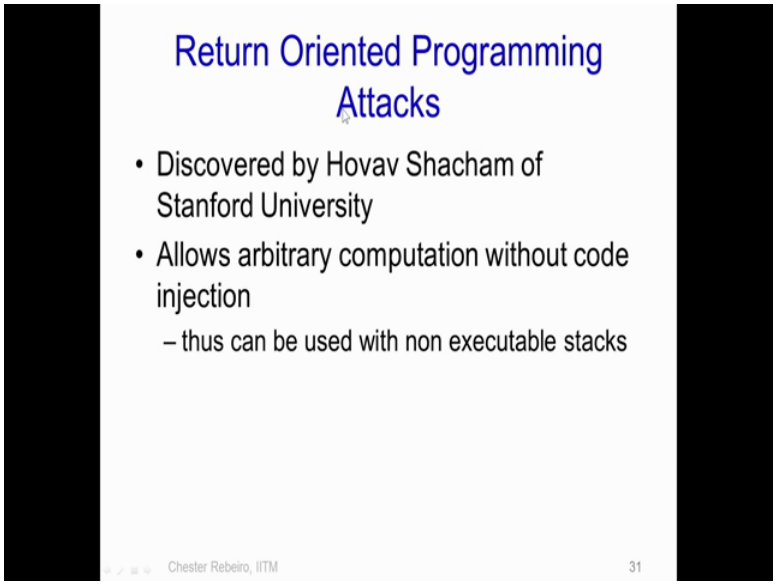
So, once this is done, when the function returns instead of returning to the valid address, it is going to cause the CPU to execute this particular system. Now during this process, also the pointer to this particular shell /bin/bash would be considered in the system call and it would result in a shell being formed. So, from the shell, the attacker could then spawn various other things and run his own programs. So, in this way, the return to libc attack works in spite of having a stack which is non-executable. So, note that we are not executing any of these instructions (mentioned in the buffer stack); we are just reading and writing to the stack while the real execution occurs in the code segment itself by the function system().

(Refer Slide Time: 10:51)



Now, the limitation of the return to libc is that it is extremely difficult to execute some arbitrary code. So, we had seen one example of how the attacker could execute a shell, but the amount that an attacker could do with the return to libc type of attack is very limited. And therefore, the attacks over the period of time have evolved to something more stronger.

(Refer Slide Time: 12:24)



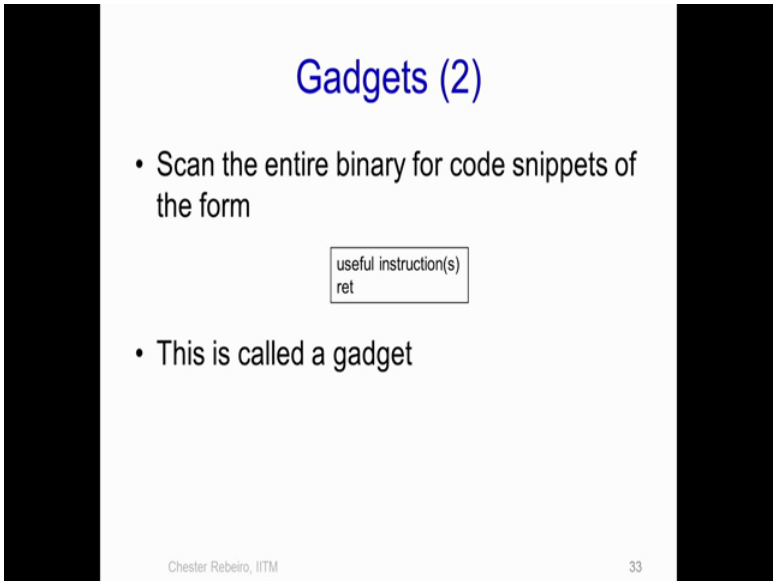
And more recently, there is something known as the Return Oriented Programming Attacks. So, this is one of the most powerful attacks that are known which utilize buffer overflows. So, this is also applicable for systems which have non-executable stack. The return oriented program or also for short known as the ROP attack was discover by Hovav Shacham in the Stanford University and it allows any arbitrary computation or any arbitrary code to be injected into the program in spite of having a non-executable stack. So let us see with a very small example how this thing works.

(Refer Slide Time: 12:13)

So let us say that the code that the attacker wants to execute is given by these set of assembly lines. So, essentially if the attacker manages to execute these assembly codes, then his job is done, he will be able to run whatever exploit he wants. Now what the ROP attack does is use something known as the Gadget.

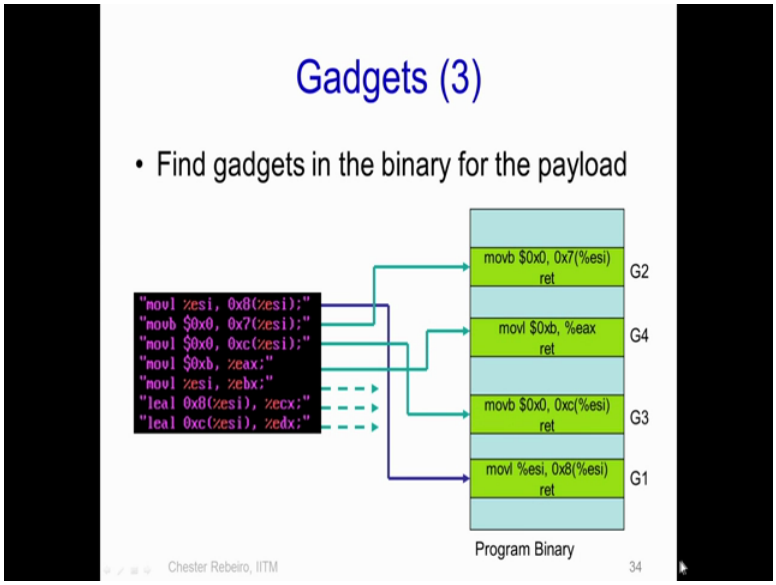
Now gadgets essentially mean splitting this particular code or the payload as it is called, into small components which are known as Gadgets. So, it has been shown that a variety of different payloads could be executed just by using gadgets, and therefore we could have a variety of different exploit codes that have been used by the attacker. So let us see what a gadget is?

(Refer Slide Time: 13:12)



So essentially a gadget is some useful instruction, and useful instruction over here it means that it is one of these instructions in the payload that needs to be executed as part of the exploit followed by a 'ret' - a return (mentioned in above slide image). So, ret as you know in the Intel instruction set it means the return from the particular function. So, this very simple thing is - what is a gadget? So now, what the attacker needs to do is corresponding to the payload that he wants to execute, he needs to scan the entire binary code of the executable in order to find such useful gadgets.

(Refer Slide Time: 14:00)



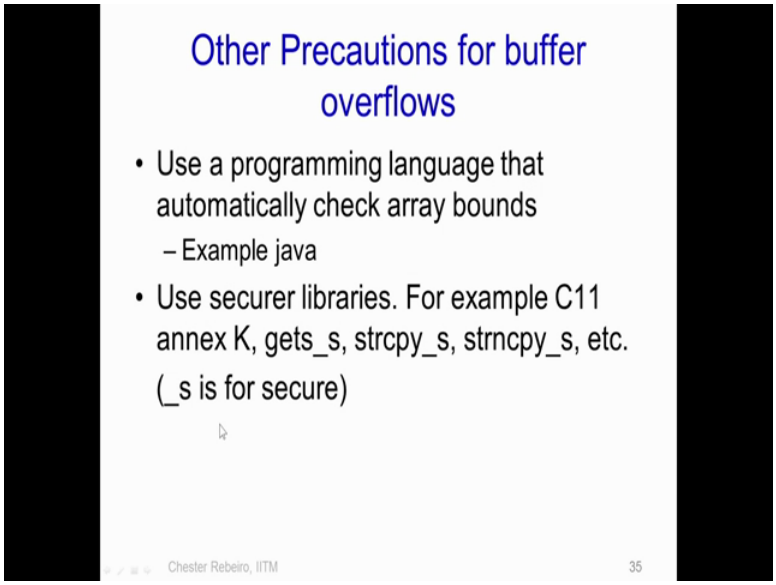
So, for instance, if we take this particular payload (refer above slide image), the attacker may find that somewhere in the programs binary that is among all the instructions that are present, there is this particular gadget (i.e movl %esi, x8(%esi) ret) present which does almost the same thing what is required that is this particular instruction in the payload is movl %esi, x8(%esi); and this exact same thing is present over here (G1 in program binary mentioned above).

Similarly, what the attacker would do is he would scan the other parts of the program binary in order to find more such gadgets. So the second line in the payload is present in the gadget 2 (refer slide time 14:00); the third line is present here (above G1) and so on. So, essentially what he would do is force this payload to execute by using gadgets. So, what he is going to overflow the stack with is a chain of gadgets, so essentially the stack is going to contain G 1 then G 2, G 3 and G 4.

So these gadgets are or the addresses to these gadgets are organized in such a way that when the first valid return address is met instead of finding the valid return address, the address of gadget 1 is found and as a result this instruction (i.e movl %esi, 0x8(%esi)) corresponding to gadget 1 is executed. And then there is a return and the stack is arranged in such a way that gadget 2 will then execute, then gadget 3, gadget 4 and so

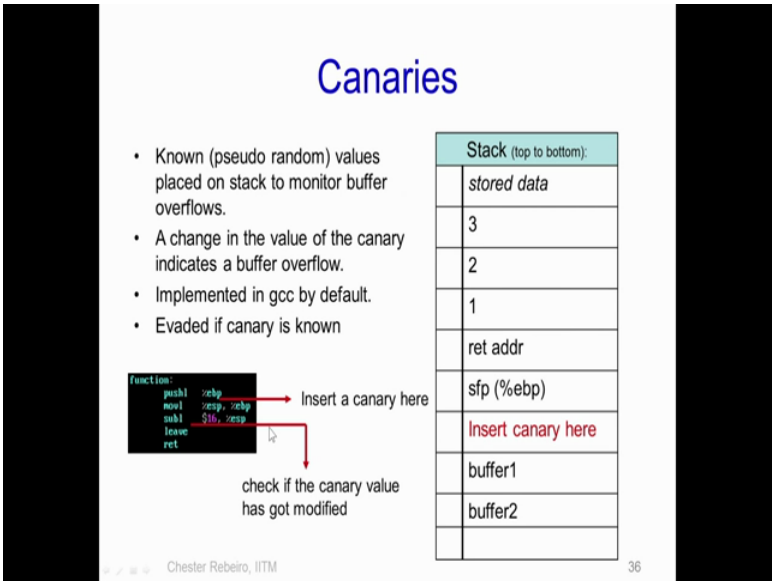
on. So, in total although the instructions are not in contiguous locations what the attacker has managed to do is he has managed to execute all these instructions (payload instructions). So, in this way, he is able to execute his payload and it has been shown that a large set of such gadgets are feasible in programs.

(Refer Slide Time: 16:22)



So, other precautions for buffer overflows, is to use a programming language such as java which automatically checks array bounds. So, this will ensure that no array is accessed out of its limits. Another way is to use more secure libraries. For example, the C 11 specification annex K, specifies these securer libraries to be used (mentioned in above slide image). So, for example, the gets_s, strcpy_s, strcnpy_s, so all these have the same functionality as the standard functions that we use, but these functions are secure and would prevent buffer overflows.

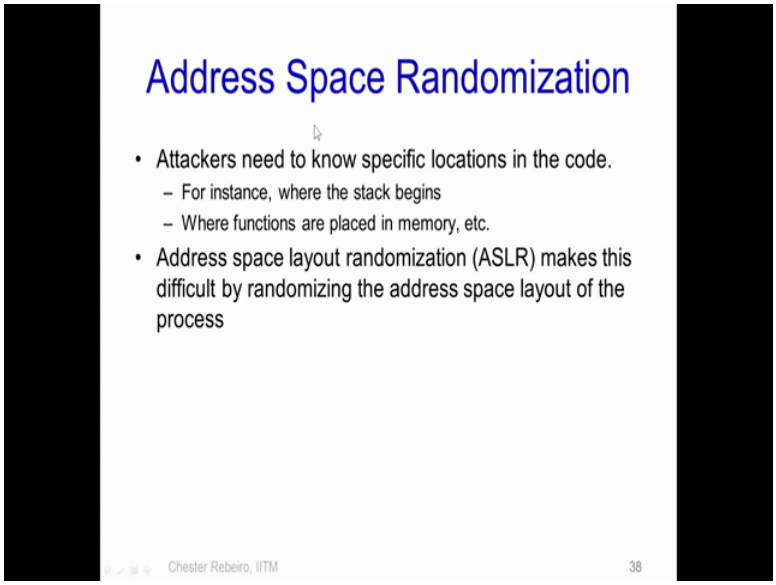
(Refer Slide Time: 17:14)



Another popular use to prevent buffer overflows is by the use of Canaries. So, a canary is a known pseudo random value which is placed onto the stack in order to detect buffer overflows. What is done is that at the start of the function a canary is inserted onto the stack; push some canary value on to the stack as shown over here (inserted canary in function mentioned in above image). So, in addition to the parameters, the return address, the frame pointer, we have now a canary also (Stack mentioned in above slide). And just before returning or leaving the function, the canary is checked to find out whether it is modified.

Now if a buffer overflow occurred then as we know the buffer overflow would modify the addresses in the stack, and as a result the canary value would be modified, and therefore, we would be able to detect a change in the canary value if a buffer overflow occurred. And therefore, we will be able to perhaps stop the program. So, these days in recent versions of the gcc compiler, such canaries are implemented by default. So, this being said the entire use of canary is evaded if the canary value is known, that is if the attacker manages to know what the canary value is used then he could just change or set this value of canary in such a way that this canary is not changed at all and therefore, its use is limited.

(Refer Slide Time: 18:56)



Another way to prevent this particular attack something known as the Address Space Randomization or ASLR, in this particular counter measure technique it uses the fact that the attackers need to know specific locations in the code. In the return to libc attack for instance, the attacker needed to know where the function F 1 or in our example where the function system() was located in the program space.

Now if we had ASLR enabled then the layout of the address space is randomized therefore, the attacker would find it difficult to determine where exactly the function is present that needs to be exploited. In other words, the attacker would find it difficult to find out where the function system() is present in the address space. Thus it would make the attack much more difficult.

Thank you.