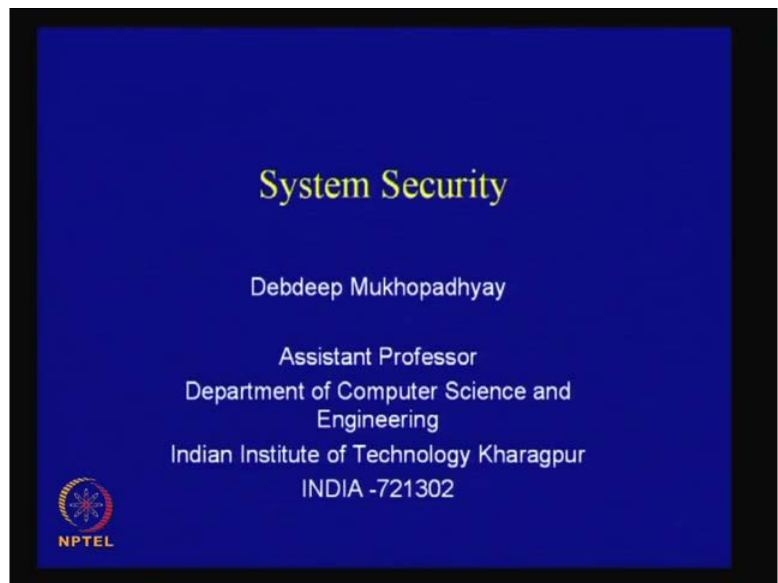
Cryptography and Network Security
Prof. D. Mukhopadhyay
Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur
Module No. # 01
Lecture No. # 39
System Security

Welcome to this very important topic on system security. So, we have seen in the previous portions of this course, various discussions on cryptographic techniques, cryptographic principles, their applications to network security and network protocols. But, here is a very important concern, that is finally after doing all these things, whether we are really secured or not.

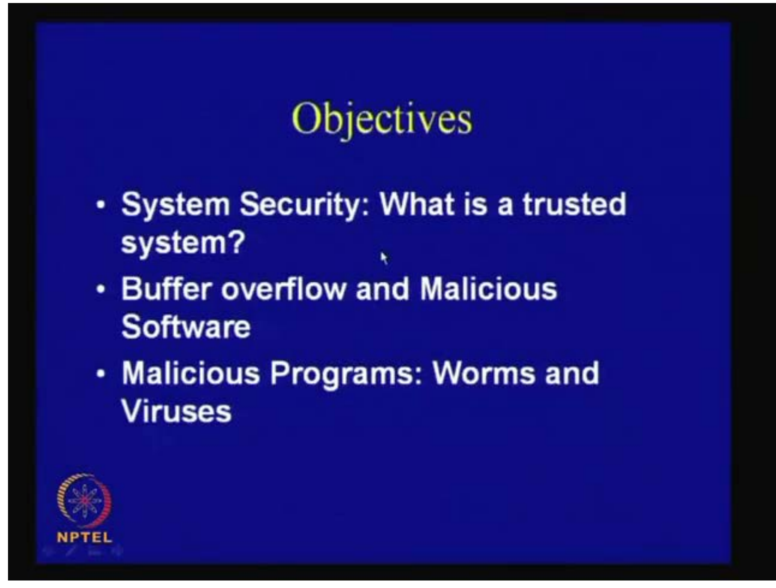
(Refer Slide Time: 00:23)



So, whether do we have trust in the machines that we are using for our communications? So, we shall see that there is a difference between actually having strong cryptographic principles and between actually having software, I mean see I mean really secured engineering.

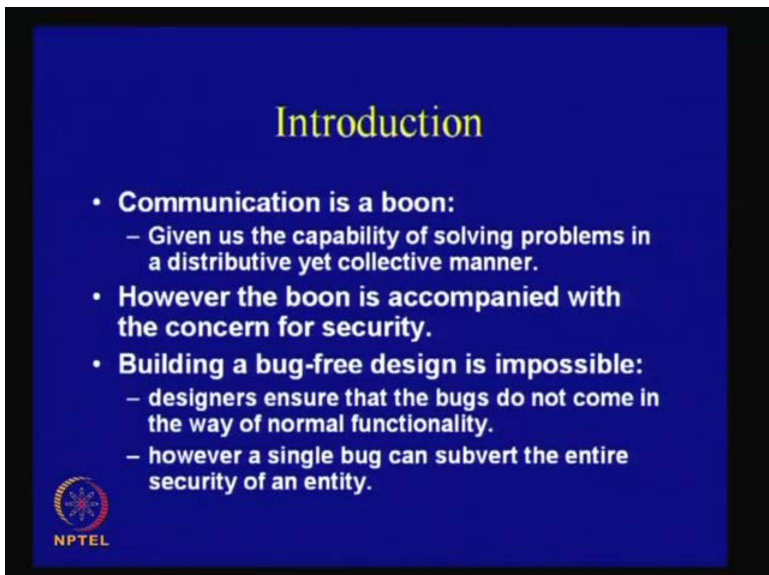
So, that brings us actually something which is probably provided by this important topic on system security.

(Refer Slide Time: 01:11)



So, let us see what are the topics of discussions. So, we shall first of all try to address this, what is a trusted system that is, what is really meant in the trusted trust in systems. We shall in this context discuss about buffer overflows and malicious software's and about malicious programs, which are very I mean commonly known as worms and viruses. There are various malicious programs, but we shall actually concentrate on the worms and viruses and discuss about the differences.

(Refer Slide Time: 01:36)



So, to start with we know that, the communication is a boon; that is when we actually have got a multiple, when you have when you have got a multiple networks right, where we have distributed networks, where we where we actually spread across the networks.

The importance or the interesting thing is about we know, how to solve problems in a distributed (( )) collective manner, but this boon actually comes with added problem that we also have to enforce security, because there is a problem of trust like there are two users who are actually I mean, somehow connected with a actual network, but they do not trust each other, right.

So, you can imagine like with the amount of complications which have they are grown up in the systems, building a bug free design is actually almost impossible right. So, even if you want to make a complex multiply or may be a complex engine for cryptographic purposes, it is actually hugely complicated, right. So, therefore, there is you cannot actually make it completely bug free.

Now, while for most of the applications, you can actually like when it is just performance related, you can actually leave with some other bugs because, if there are some bugs it may happen that those bugs are never triggered. But, in the context of security, if you have a slightest amount of bugs then, those bugs can be actually exploited for attacks and therefore, the entire security can be compromised.

So, therefore, you have bugs for most of the purposes can be actually tolerated, can be lived with, but whereas security is concerned, bugs can be absolutely fatter. So, we will see some of these in our descriptions and discussions.

(Refer Slide Time: 03:08)

Description of the System

- Proper description of a system and its boundary is extremely important.
- A system is a vague entity that comprises the totality of the computing and the communication environment.
- A system boundary demarcates between what is attempted to be protected (system) and what is unprotected (external world).

So, now first of all, we have to define what is meant by a system. Now, we will see that a system is actually a vague entity, really speaking. So, it is actually comprise the total of all the computing things like the computing, it could be likes networks, it could be like any, it could be like your computers, it could be your hardware, so it could be the software and the hardware, which are used for computation purposes, and also along with there is a communication environment that means, the network. So, the entire thing is part of actually what we term as the system.

So, now, what is a system boundary? System boundary is something which actually demarcates this system, that is what I am trying to protect with the external world, which is actually beyond the scope of my definition of my system, ok.

So, that that the boundary is actually, what demarcates between these two. Now, what are the components inside a system? So, there are actually three, you can where do we say that, there are three important components. One is actually which is security relevant, security relevant means that, which are absolutely important that they are secured.

So, it could be the operating system, it could be the hardware, which is actually imperative that they are they have to be secured.

(Refer Slide Time: 04:25)

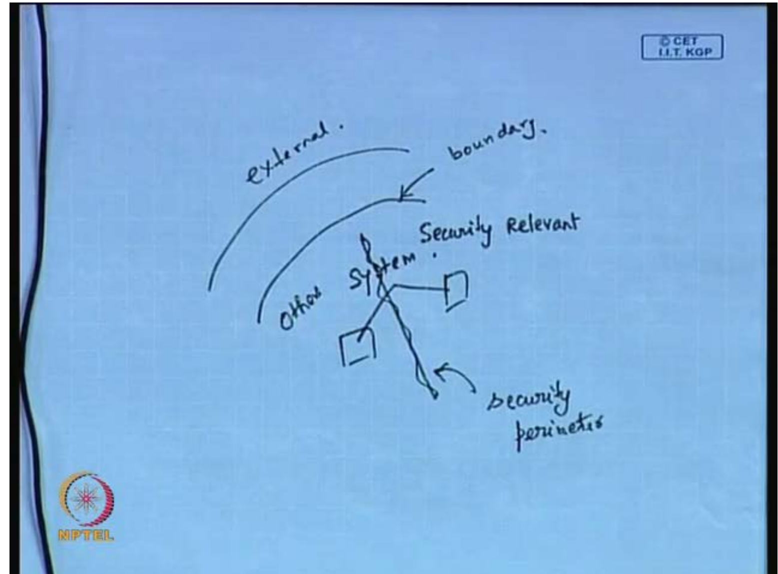
Components inside the system

- Security Relevant: These components are crucial to the security, eg.: OS and hardware.
- Others: These are objects that the system controls and protects, eg.: programs, data, terminals, modems.
- Security Perimeter: Line of demarcation between security relevant and other components.

NPTEL

The other thing could be like could be like the programs, data terminals and modems. So, these are actually something, which I am trying to protect, you can say. So, note that this is program and not process, so that means, it is something which is not here running. So, therefore, this could be like what I am trying to protect and a security perimeter actually is such kind of line of demarcation between the security relevant systems and the other systems. So, you can think that, very widely you have got the external world, we have the external world.

(Refer Slide Time: 04:56)



We have got the system, that is what your essentially what you call as like you may have your computers and it may have your computers separated by networks and then this is what is known as the boundary. So, this is your system boundary and even in your system, there are actually we can broadly divide into two parts; one which is actually very much security, when you called as security relevant right.

So, it could comprise of the operating system in the hardware and these are what we actually try to protect the others and in between this, we have something which is known as the security perimeter. So, this is you can kind of pictorial visualize the entire scenario for systems ok.

(Refer Slide Time: 05:54)

A common misconception

- "Encryption provides system security":
  - encryption provides data confidentiality.
  - for system security we need trust in the security related components.
- If the OS is compromised, can we trust the result of an encryption program running on it?
  - hence other methods and even principles and policies are necessary.

NPTEL

So, now there are various misconceptions which actually are there with us, like this particular statement like encryption provides system security. So, you see that why we have seen that encryption provides data confidentiality and other things like what we have seen previously in previous discussions, for system security what we need is actually trust in the security related components.

We can imagine like, if there is an operating system which is compromised, can we trust the result of the encryption programs we are running on that? We cannot trust right. So, therefore, other methods and principles or policies are absolutely necessary in order to enforce what we know as system security. So, encryption alone does not actually give them. So, therefore, we have to actually understand some of the principles which will provide us an end to end security.

Because finally, at the end of the day, we actually have to in spite of doing all that theory and doing all, that having all that cryptographic principles and techniques in place, we also need to develop other principles which will give us an end to end security, the final security, we cannot leave a weak hole basically.

(Refer Slide Time: 06:58)

So, therefore, the concept of trust and trusted systems, so as we know that, in a trusted systems scenario typically there is a user, who is given a trusted piece of information should which could be a login or some other authentication code. So, by which it actually tries to convince the particular client or the system that it is an authenticated server. So, if you remember the (( )) protocol, so similar to that a user actually first communicates to the client, authenticates then the client communicates with the (( )) server, then communicates with the ticket granting service and then finally, communicates with the service that it wants right.

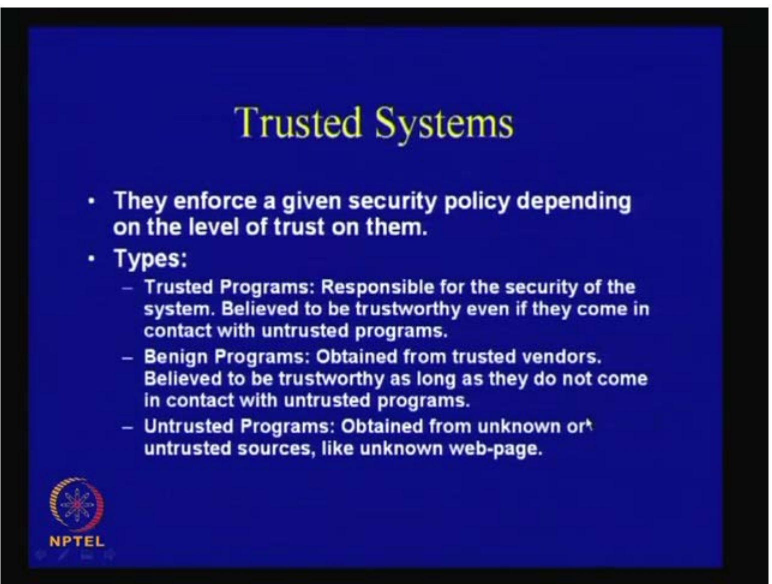
So, therefore, the user is actually understood in by the system through a login, like could be an authentication tag right. So, now, therefore, if any systems security notion what we assume is that, the user does not leak away this particular login. Of course, if we if that is leaked out then, we cannot actually adopt any principle to get on the security, right.

So, the assumption is that, it is expected that the user does not give away this valuable information and the system security should actually if the if it may happen that, the user is actually deceived to manipulate the data, which leaks the secret then the security measures of the system should actually warn the user against such accidental or unintentional leakage right.

So, if the leakage is actually intentional then, system security cannot do anything right, but if the leakage is unintentional or accidental, then the system security principles or technique should actually warn the user against such a leakage that is the objective. So, therefore, trust is built in a system using authentication and identification, when the system recognizes a public ID could be a login name etc and what it does is, it actually associates every process with a user, ok.

So, the violation of authentication is serious and therefore, it will break the trust in a system. So, therefore, the idea is that, wherever there is a violation of authentication there was an accidental or unintentional leakage of the user secret, then the system security techniques should actually enforce so ensure that, there is a warning which is provided to the corresponding user. So, therefore, that is the way how we built trust into a system.

(Refer Slide Time: 09:19)



So, now I mean, so how do you enforce that, so therefore, they enforce a given security policy depending upon the levels of trust. So, there are various levels of trusts which exist, like there could be some which is called as a trusted program which are responsible for the security of the system and it is believed to be trustworthy, even if they come in contact with the untrusted programs ok.

So, therefore, there are certain programs which are actually believed to be always trusted like if even if they come into contact with untrusted programs, we believe that their trust is not compromised. On the other hand, we have got something which is known as benign programs. Now, these benign programs are actually obtained from trusted vendors and it is believed to be trustworthy as long as they do not come in contact with the untrusted programs ok.

So, therefore, if they come into contact with the untrusted programs, which are actually obtained from unknown or untrusted sources like may be unknown web pages, then we are not guaranteeing the security or trust of the benign programs, but the trusted programs are always trusted. So, it could be like the hardware or the essentially the corner of the system ok.

But, but on the other hand, there could be some other things like may be some say particular software, which you have actually got from somewhere. So, we believed that they are benign programs, but then they can be also subjected to attacks and could be actually made untrustworthy by some other untrusted programs ok. So, therefore, you see that there are various levels of trust which exists in our system and the policy for security depends upon the level of trust which is attributed to them right.

(Refer Slide Time: 11:04)

A Code Snippet

```
int main( ){
    int buffer[10];
    buffer[20]=5;
}
```

- Compilers will compile the above program without any error.
- This seemingly simple problem can be used by the authors of malicious softwares to trigger malicious codes.

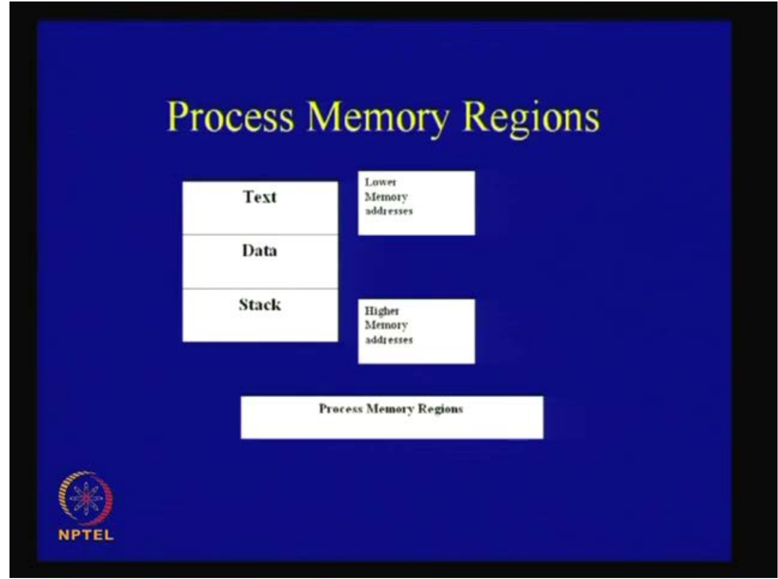
NPTEL

Simple c program, so you consider that in buffer 10, buffer 20 is equal to 5. So, these are very common c program. So, you see that a compiler will compile this above program may be without any error. Now, this seemingly simple problem can be actually used by the authors of malicious software's to trigger a malicious code ok.

So, let us see first how? So, therefore, the first thing to be observed is that, here we have got a buffer which has got a size of 10 integer variables. So, typically you can say that 10 into 4 will be so if you assume that were 4 bytes would be a 40 byte location, right allocated.

So, but when you are writing into this 20th location, we are actually not thought about this 20th location, but what we are doing is that, we are evaluating into a something which is passed which has gone passed the the the the the size of the buffer which was intended ok. So, now the first thing to be observed is that, this does not give any error. So, this will compile without any error.

(Refer Slide Time: 12:07)



So, now, in order to understand this, the the the the context we have to let us consider the process memory regions. So, typically when your process runs here, there are three important regions in the memory; one is the text, the other is the data and the other one is the stack.

Now, it generally, typically goes from the lower memory to the higher memory and the text is something which is used for many sub binary instructions and something which is hot coded. So, therefore, the idea is that you cannot write into this text region because if you write into this text region, it may end up with something like a segmentation fault and this part like the data part is actually when you used for allocating spaces by thus the static variables ok.

So, therefore, variables which are static in the nature can be actually allocated space in the data, it could be initialized, it could be uninitialized as well, but on the other hand, this stack is actually the other thing, which we will focus in our discussions is actually used to allocate space for the dynamic variables. So, let us see like, how essentially the allocation in the stack can be actually exploited for being unintentional objectives or actually in unintentional objectives ok.

(Refer Slide Time: 13:16)

Stacks

- Contiguous block of memory containing data, which grows from higher memory address to lower memory address.
- LIFO (Last In First Out)
- PUSH: insert an element to the stack top.
- POP: delete an element from stack top.
- High level languages use stacks to implement function calls.

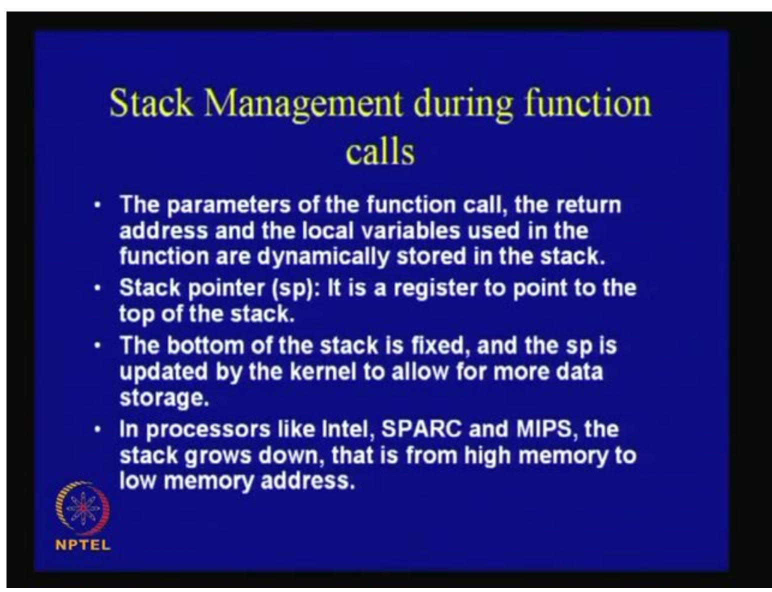
NPTEL

So, what is the stack? So, stack is actually a contiguous block of memory containing data, which grows from the higher memory address to a lower memory address. This is very typical thing like generally for our most common processes, this how the stack is being maintained that is, it grows from the high memory address to the low memory address.

Now, the concept of any stacks as we know is Last in First Out or LIFO and the two important operations in stacks are push and pop; that is when you insert an element into this stack top and when you delete from the top of the stack.

Now, high level languages use stacks to implement function calls, because whenever there is a function call, then you save here to represent variables and you can actually go into the new function call and allocate space for the new variables there and also store the return address, where you have to go back after the function has been executed, right.

(Refer Slide Time: 14:11)



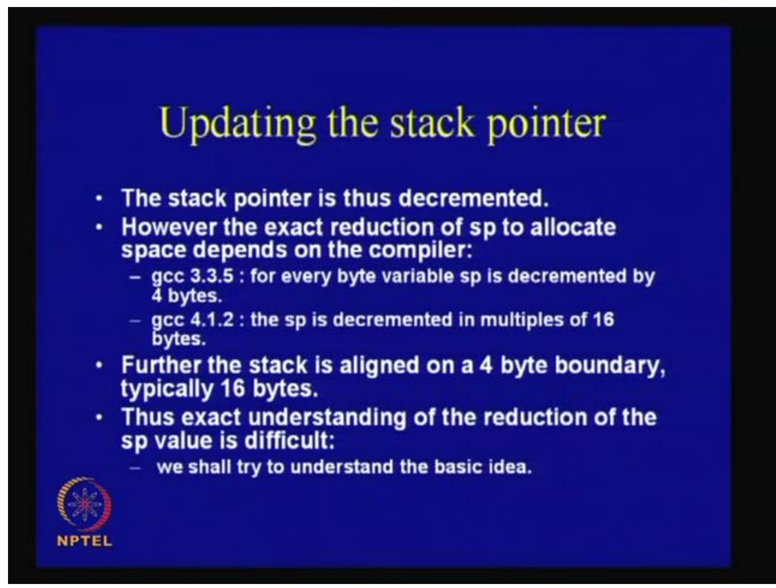
So, therefore, typically we will see that, there is a lot of stack management involved when there is a function call, right. So, the parameters of the function call, the return address and the local variables used in the function are dynamically stored in this stack. So, whenever there is a I mean I mean whenever this as the there is a function call, then the parameters of the function call, the return address that is the address where it has to return back to where the instruction points the points to and therefore, and also a local variables used in the function are dynamically stored in the stack.

Now, the stack pointer is a register, which is also commonly known as the sp register. It is a register to point to the top of the stack and the bottom of the stack is actually fixed and therefore, the sp is and sp is actually updated by the kernel to allow for more and

more data storage, that is whenever more data storage is required, we actually update the stack.

So, since the stack normally grows from the high memory to the low memory, we will actually decrement the stack to allocate space for next extra variables or extra space. So, typically in processors like Intel, SPARC and MIPS, the stack grows down that is from high memory to the low memory address.

(Refer Slide Time: 15:23)



Now, we need to update the stack pointer. So, therefore, the stack pointer is thus decremented; however, the exact reduction of the stack pointer to allocate space depends on the compiler and may vary from compiler to compiler. So, typically we have seen that in our gcc 3.3.5 for every byte variable, sp is decremented by 4 bytes whereas, for gcc 4.1.2 the stack point is decremented in multiples of 16 bytes.

So, these are the some observations that we can make if you run rather see the gcc compilation and see the corresponding stack management. So, further the stack is aligned on a 4 byte boundary, typically 16 bytes and thus exact understanding of the reduction of the stack pointer value may be quite difficult; however, we shall try to understand some basic points, basic ideas ok.

So, therefore, if we are not necessarily hold on every system, but we will try to understand the basic principle behind stack based overflows or buffer overflows.

(Refer Slide Time: 16:22)

Example

```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
}
void main() {
  function(1,2,3);
}
```

In case of gcc 3.3.5, sp is decremented by 20 bytes (buffer1 needing 8 bytes, buffer2 12 bytes)

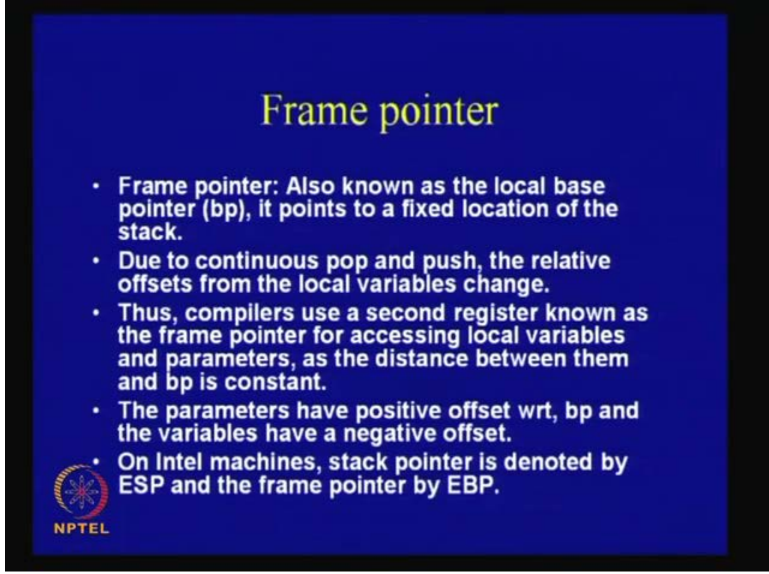
In case of gcc 4.1.2, sp is decremented by 16 bytes.

NPTEL

So, again this is a classic example. So, we can consider like there are two character buffers, buffer 1 and buffer 2 of size 5 and 10 and there is a function which is like passes 1, 2 and 3 as variables to this function.

So, in case of gcc 3.3.5, we have seen that the stack pointer decremented by 20 bytes because buffer 1 we need 8 byte and buffer 2 will need 12 bytes. And in case of this gcc we have seen, this version of gcc we have seen, there is stack pointer gives decremented by 16 bytes. So, it could be like because 5 plus 10 is 15 and the next multiple of 16 is 16.

(Refer Slide Time: 17:02)

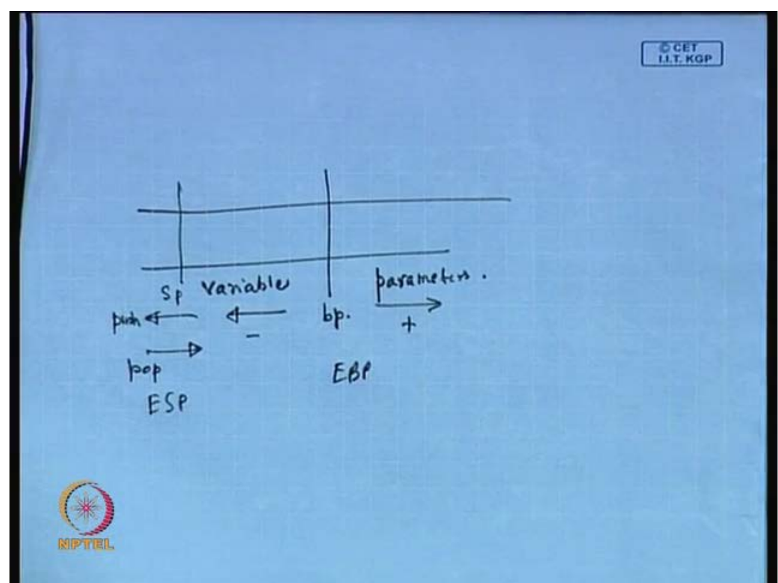


So, the other important pointer which is maintained is known as the frame pointer. So, also known as the local base pointer and it is actually it points to a fixed location of the stack. Now, due to the continuous pop and push what happens is that, the relative offsets from the local variables change and therefore, the compilers use a second register known as the frame pointer for accessing local variables and parameters as a distance between them and the base pointer is constant, ok.

So, therefore, the the frame pointer is something like the base pointers. So, whenever there is a function call, then the then the base, the current stack pointer has been updated as a base pointer and the relative distance between the base pointer and a variable is not changing is not variable, while the relative distance between a variable and the stack pointer gets continuously updated because, whenever we are doing a push or we are doing a pop, the stack pointer gets incremented or decremented, right.

So, the parameters have got positive offsets with respect to the base pointer or the frame pointer and the variables have got a negative offset. So, the parameters or the function have got a positive offset and these variables have got a negative offset. So, you can visualize like in a stack kind of scenario, so this could be that a base pointer.

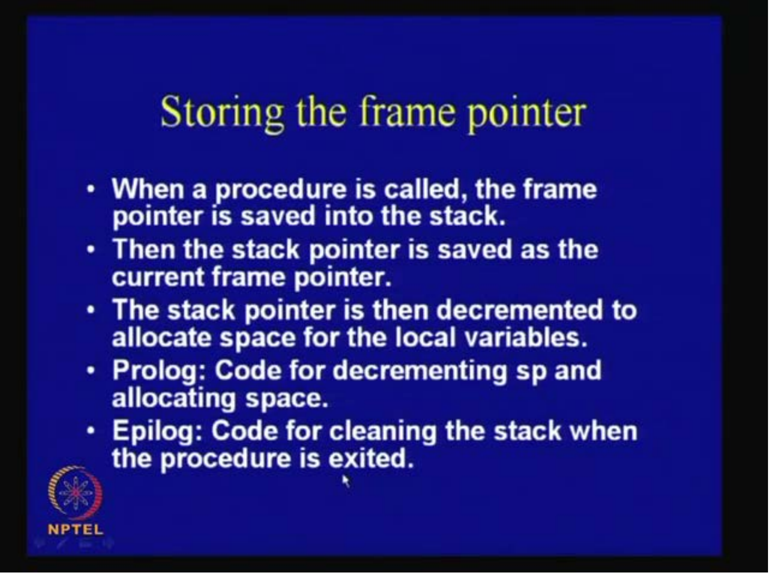
(Refer Slide Time: 18:22)



And here, the parameters are typically on this site whereas, here variables are typically on this negative offset, ok.

So, therefore, this is the negative offset and this is the positive offset that we are talking about and the stack pointer is continuously updated, that is depending upon your variable, your stack pointer moves either this side or moves either this side. So, whether depends upon whether you are doing a push, when you are doing a push it moves this side, when you are doing a pop, it moves in the other direction. On intermissions, this stack pointer is often denoted by this symbol ESP and the frame pointer is denoted by a symbol EBP ok.

(Refer Slide Time: 19:11)



So, how do you store the frame pointer? Now, when a procedure is called, the frame pointer is saved into the stack, then the stack pointer is saved as the current frame pointer and the stack pointer is then decremented to allocate space for the local variables, these are commonly known as the process called as prolog and epilog. So, the prolog is the code for decrementing the stack pointer allocating a space and epilog is the state whether code for cleaning the stack, when the procedure is exited. So, therefore, these are the two opposite operations which have been done.

(Refer Slide Time: 19:43)

An Example

```
void function(int a, int b, int c) {
  char buffer1[5];
  char buffer2[10];
}
void main() {
  function(1,2,3);
}
```

Lets generate the assembly code to see the stack management.

gcc –S –o ex1.s ex1.c

So, let us consider an example which is character buffer 1 5 and character buffer 2 10 and again this is a function. So, let us consider, I have see the assembly code which has been generated by using this particular switch to see the corresponding stack management ok.

So, therefore, I am just giving you some snapshots like the immediate so you can see that, the first thing which is to be noted here is that, these things need to be stored, right the variable the parameters are 1, 2 and 3. So, therefore, what is what we see is in our in my machine, when we have obtain the corresponding assembly, then first 3 is actually you have stored, then 2 is stored and then 1 is stored. So, this is again relative with respect to the current stack pointer notation.

So, the immediate values indicated by so these are actually indicated by this dollar symbols. So, these are the immediate values, these are saved and the numbers before this are actually indicating the offsets, the corresponding offsets and this stands for the address of the stack pointer. So, therefore, this actually stands for the stack pointer address and this stands for the corresponding offset with respect to the stack pointer (Refer Slide Time: 20:45).

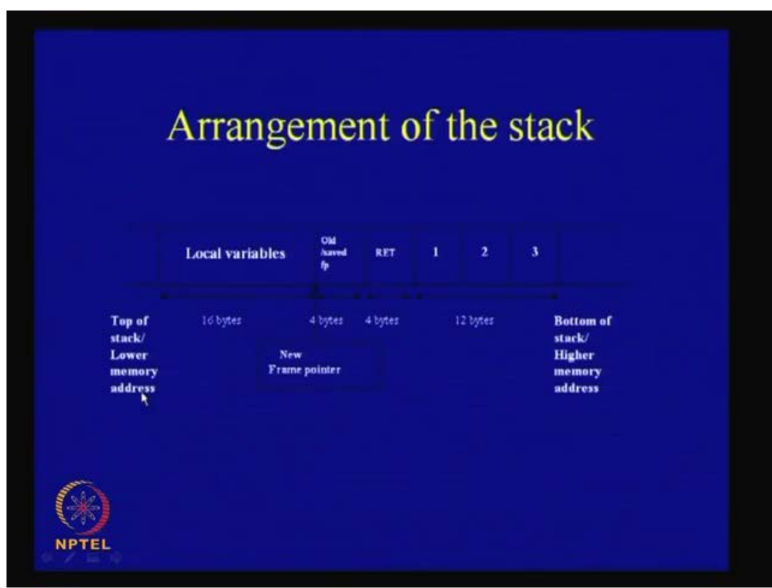
So, therefore, this was an offset of 8, this was an offset of 4 bytes and this was an this is the place where you has pushing it and in this case, we see movl has been used, you can

also think this to be like a push. And the instruction call will actually push the instruction pointer into the stack, the saved instruction pointer is a return address or RET to which the program counter jumps after the call is executed.

So, after the call is executed, it has to go back to the old return address; therefore, the instruction pointer has to be stored. So, therefore, that is the first thing which is being stored, after that what we do is that, we update the frame pointer. So, the frame pointer EBP is pushed into the stack and so, therefore, this is being done here. So, you can see that pushl EBP means, the corresponding frame pointer is pushed into and then we save this current stack pointer has the frame pointer and then we actually adjust the stack pointer for so in this case, again 16 bytes have been allocated.

So, therefore, depending upon the variable size, since there we have a total we have a character size of 15, so that means, 15 bytes are necessary. So, we allocate space for 16 bytes for both the buffers. So, therefore, the current stack pointer is saved as the new frame pointer. So, and the stack pointer is decremented by 16 bytes to allocate space for the character buffers of total size 15 bytes.

(Refer Slide Time: 22:17)



So, therefore, cryptographically these are if they look like (Refer Slide Time: 22:21). So, like that this 3 2 1 and this is the 3 2 1 and this is the corresponding return address, this is the old or saved frame pointer, which has been saved previously and the new frame

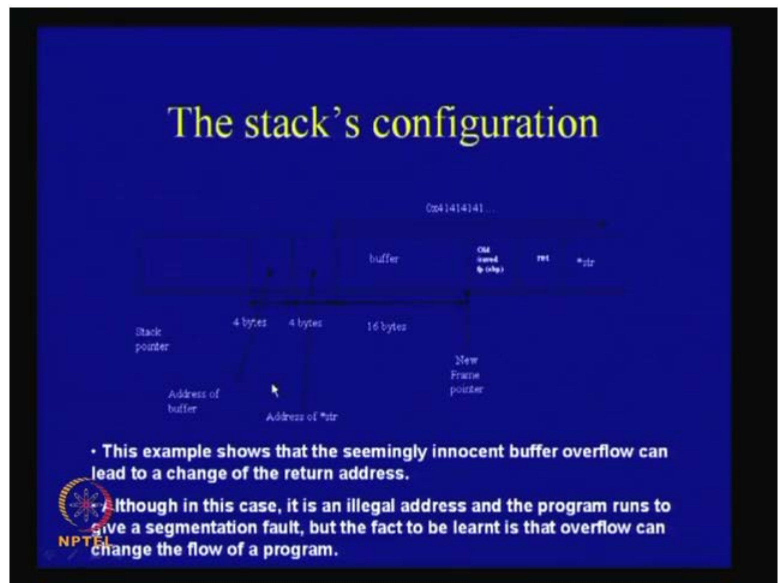
pointer actually points at this and these are the local variables and this is the type of the stack ok.

So, therefore, this is the space where actually the local variables are actually stored or the values of the local variables are stored. So, now, you see that this is the broad arrangement of the stack and therefore, we will see that what happens when you do overwrite into this stack?

So, in order to do that in order to do an overflow, so let us see like, let us consider this example, where there is a character buffer of c I 16 and what we do is that, we use a stream copy. So, therefore, this function actually takes a parameter of pointer to a string and it will actually just paste this or copy this into this buffer. Now, when we pass this particular string or other this particular string to this function, instead of this size of 16 bytes let us pass a larger string.

So, let us pass a string of size 256 characters and let us fill it with ASCII value of a, let us write try to evaluate into this buffer.

(Refer Slide Time: 23: 41)



So, now, what happens is that based upon our previous discussion, this is the space which is there for the buffer, right. This is the space means, this is the corresponding size of the buffer that is 16 bytes along with there is the address of the str also because you

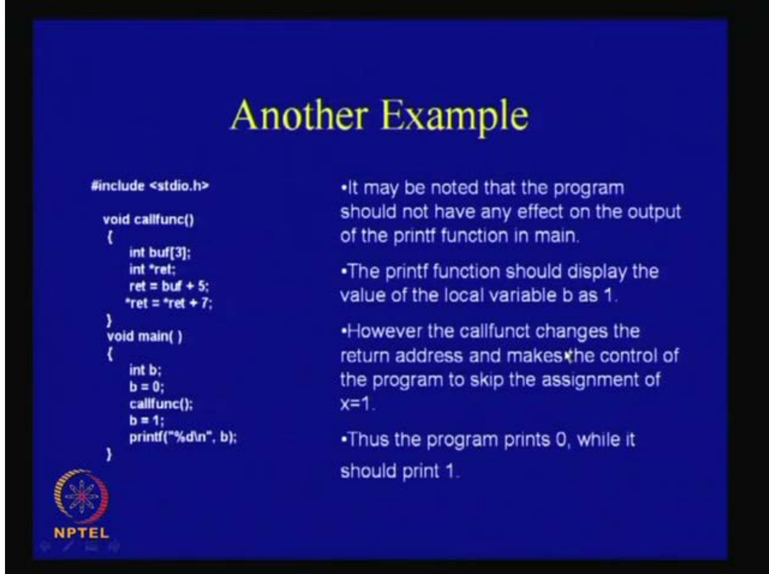
have seen that, we have used this string copy so, there is also again another function call (Refer Slide Time: 23:45).

So, which means that, there is a pointer or the base pointer of the string and there is a base pointer of the buffer also. So, therefore, these are also there. So, this is the address of the of the str and this is an address of the buffer. So, again this and this comprise of 4 bytes and 4 bytes each and this is the updated point, where the stack point has been updated, ok.

So, therefore, now this example shows that, when we are actually when we start writing in this ASCII value of a, which is actually 4 1. So, then 4 1 actually writes from this point and continues and actually goes, pass the end of the buffer and may continue to n modify or code of this return address, right.

So, therefore, this example shows that the seemingly innocent buffer overflow can actually lead to a change of the return address. The return address can be using this kind of overflow techniques. So, although in this case, it is an illegal address and may happen that a program gives segmentation fault, but the fact is that this (( )) of return address is can be used to actually change the flow of existing program ok.

(Refer Slide Time: 25: 08)



So, let us now, I mean just see one more example where it may happen that, the program actually do not end up in a segmentation fault and actually end up in doing something

what is not intended ok. So, therefore, consider this simple problem, seemingly simply problem, that you have got a main function, where you just update a local variable b equal to 0; call a function which actually does not seemingly tampered b and then you again may b equal to 1 and then print f.

So, therefore, it does not matter whatever you do here, since b is 1 you expect that b should be printed as 1 right, but what may happen is that, it may happen that this does not end up in getting what is intended. So, therefore, it may be noted that the program should not have any effect on the output of the print f function in main; the print f should actually display the value of the local variable b as 1. However, the call function changes the return address and makes the control of the program to skip the assignment of x equal to 1.

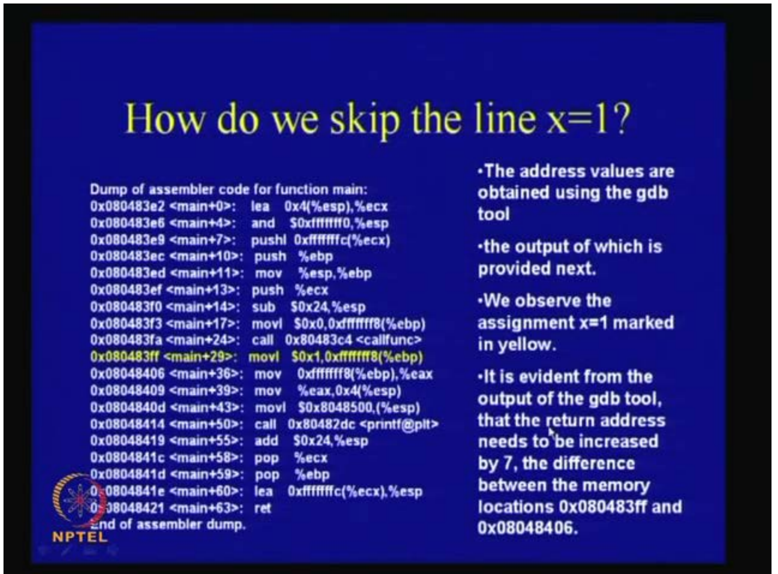
So, it may happen that, so, sorry this should be actually b equal to 1. So, therefore, it may happen that the call function because what we expect is that once this function is called it actually returns and comes back and executes the this line b equal to 1. So, what may happen is that, because of an existing buffer of flow in this function, the corresponding return address gets changed and gets changed in such a such a fashion that, this particular line is skipped and we come to this print, where we print actually b equal to 0 which is the old value of p, ok.

So, now how is this done like how is the return address skipped, I mean how is this particular line skips? In order to understand that, let us again see the stack in the snapshot of the stack. So, we will see here that these are the 12 bytes, which is being allocated here, this is the corresponding return address 4 bytes and there are 4 bytes here. So, we can see immediately that 12 plus 4 is 16 plus 4 is 20. So, depending I mean relating to this, you have got a you have got an offset a positive offset of 20 bytes.

So, therefore, that buffer plus 20 actually will point to the content in the stack, where the corresponding return address has been stored. So, therefore, you can see that program now, so you’re actually jump by 5 integer locations. So, there are 5 integer means 5 into 4 is 20 and therefore, we come to the place where the return address has been written and instead modify the content of the return address, where the content where the pointer return points 2. So, star at ret is the corresponding value there and we update it or increment it by 7, ok.

So, somehow and so, let us forget right now why this 7, but what is important is that, we are actually changing the content of the corresponding return value. So, whatever is the return value here, the return address here is actually incremented by 7.

(Refer Slide Time: 28.00)

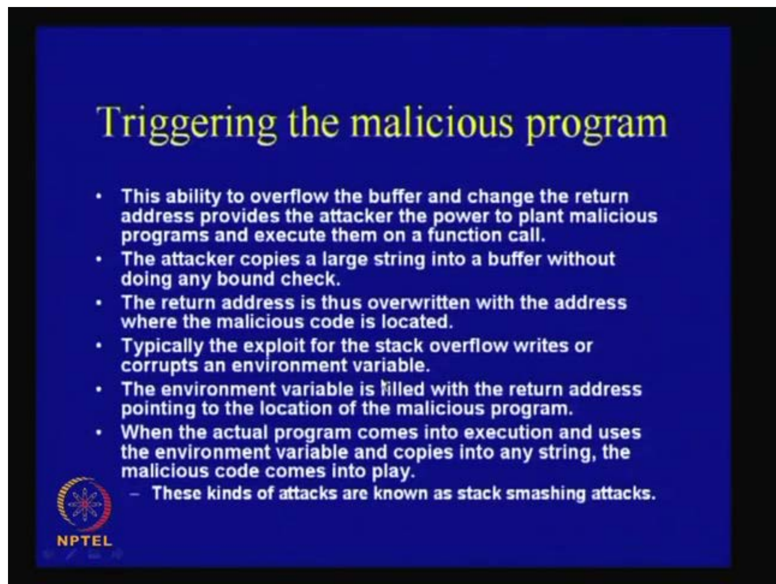


So, now, why 7 for that this is just a simple snapshot of how which of this lines and this has been generate by the gdp 2. And we see that this is the line, which is actually what which actually does the assignment of b equal to 1. So, we want to skip this line.

So, therefore, in order to skip this line, we have actually to increment this pointer or the return value because the return value presently should be this and we have to increment it by 7 places because that is a difference between this memory location and this memory location. So, it is evident from these two that, actually the return address needs to be incremented by a step of 7. Once we increment it by 7, then this line gets skipped and we come to the next line which actually does not do the do the do the assignment of b equal to 1.

So, therefore, we are actually skipping this yellow line and we therefore, get the value as b gets printed as 0 and not 1 because the assignment of b equal to 1was never done.

(Refer Slide Time: 29: 06)



Now, what we see or rather what we learn from this simple example is that, this ability to overflow the buffer and change the return address provides the attacker, the power to plant malicious programs and execute them on a function call.

Now, the attacker it may happen like, we can consider like the attacker copies a large string into a buffer without doing any bound check, the return address is thus overwritten with the address where the malicious code is located. So, therefore, what what an attacker can do is that, update the return address, with the address where a malicious code is being kept. And what may happen is that, this exploit can be used or whether they they exploit for the stack overflow. So, what what what it typically does is that, the the exploit for the stack overflow writes or corrupts an environment variable.

Now, the environment variable is filled with the return address which actually points to the location of the malicious program. Now, when the actual program comes in to execution and uses the environment variable and copies into any into any string the malicious code comes into play. Now, these kinds of attacks are very commonly known as the stack smashing attacks ok.

So, therefore, what is the idea? The idea is as follows, that is you have got you have got you want to essentially to change the return address to some address where a malicious program has been planted, right. So, in order to do that, we actually what what the

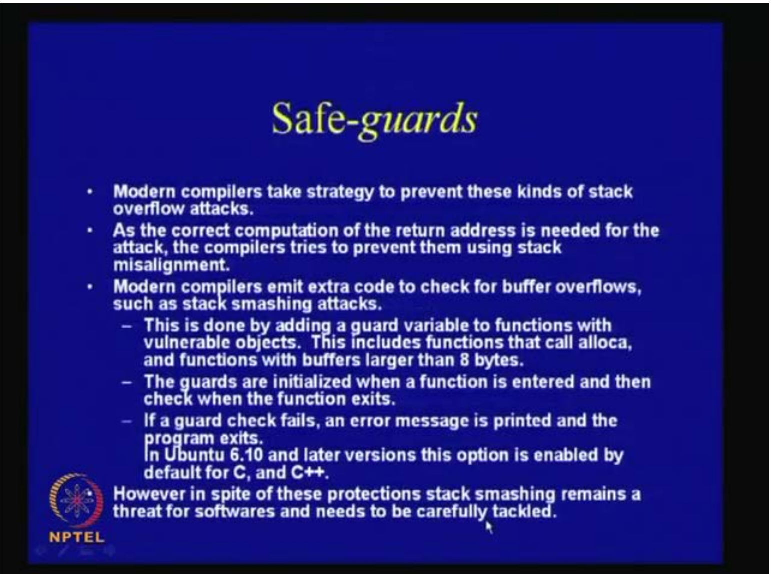
attacker can do is that, it can change some environment variable which is being used by that by a particular program.

Now, the environment variable is modified in such a way that, it is actually modified a modified to contain, or it is overwritten to contain the address of the malicious program. Now, when the program reads this environment variable what may happen is that, the string corresponding to the environment variable gets and sits into the return address and sees the environment variable contents the address of the malicious program.

Now, the return address is being changed to somewhere where the malicious program has been kept. And therefore, when we on a particular program which is suppose, may be do something some objective or achieve some objective, it may end up in actually triggering a malicious program.

So, the malicious program could be a virus, it could be a worm or some other some other program which is not intentional right. So, these kinds of attacks are something which is known as the stack smashing attacks and can be exploited for attacks.

(Refer Slide Time: 31: 29)



Now, recently there are several safeguarded modern compilers do take strategies to prevent these kinds of attacks. The correct computation of the return address is needed for the attack and therefore, what the compiler tries to do is that, it tries to prevent them by using various things like stack misalignments and things like that. So, therefore,

modern compilers emit extra codes to check for buffer overflows such as stack smashing attacks. So, it could be like, it actually adds a guard variable to functions with variable with vulnerable objects.

So, this includes functions they are actually so if there is a function like, where you are using this alloca; that means, you are allocating space to dynamic variables or if you are using buffers with a larger than 8 bytes then, these guards coming to play. Now, when this guards are initialized, whenever there is a function is entered and checks when the function is exited and therefore, these guards and if this guard check fails, then the program actually does not continue and in the actually passes an error message and the program may exits, ok.

In the new Linux environments and later versions, I mean versions like which are presently available, these are actually kind a enabled by default for c and c plus plus. However, in spite of this production stills stack smashing remains a threat for software's and needs to be kept in mind when we are writing programs.

So, the typical idea is that, when we are using functions like say str cpy and which function calls which actually does not do any bound checking. So, when we are writing programs for security, it is advice to actually instead of writing them, we actually use function calls which takes care of the bounds.

So, what we do is that, we actually replace like str cpy by things like str and cpy which actually takes care of the bounds. So, therefore, so idea is that, if replace your unbounded implementations with bounded implementations; even if you are doing so, you have to write or emit explicit codes, which will take care of the bounds, ok.

So, there is a kind of thing which is to be kept in mind, when we are writing program for security purposes. Otherwise what may happen is that, we may end up in writing a nice encryption algorithm that may end up in doing still in subjected to attacks which actually do not target the algorithm, but target our implementation. So, therefore, we have already come to the topic of malicious programs because, we have seen one way in which malicious programs can be triggered.

(Refer Slide Time: 33:58)

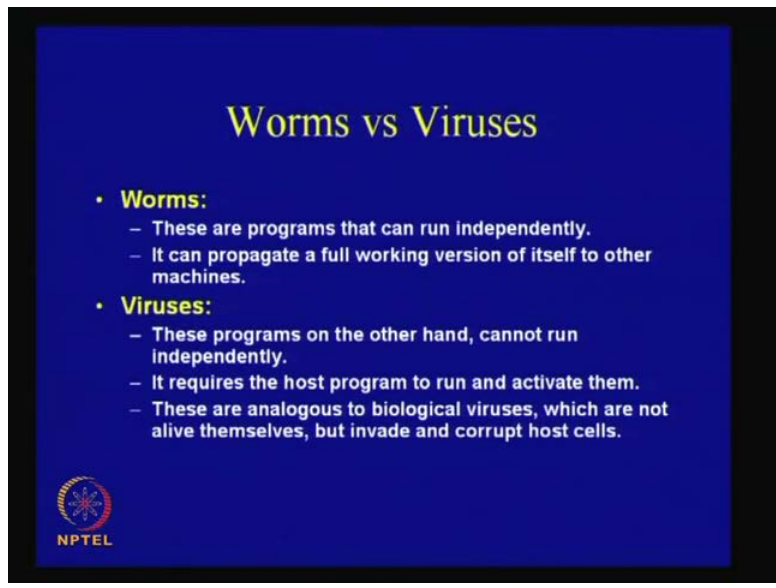
Malicious Programs

- Malicious programs, as already clarified, are those programs which try to subvert the expected operation of secured and benign codes.
- The two most commonly known categories of malicious programs are worms and viruses.

NPTEL

So, now, what are malicious programs? They are actually they are actually programs which try to subvert the expected operation of secured and benign codes. The two most commonly known categories of malicious programs are worms and viruses.

(Refer Slide Time: 34:22)

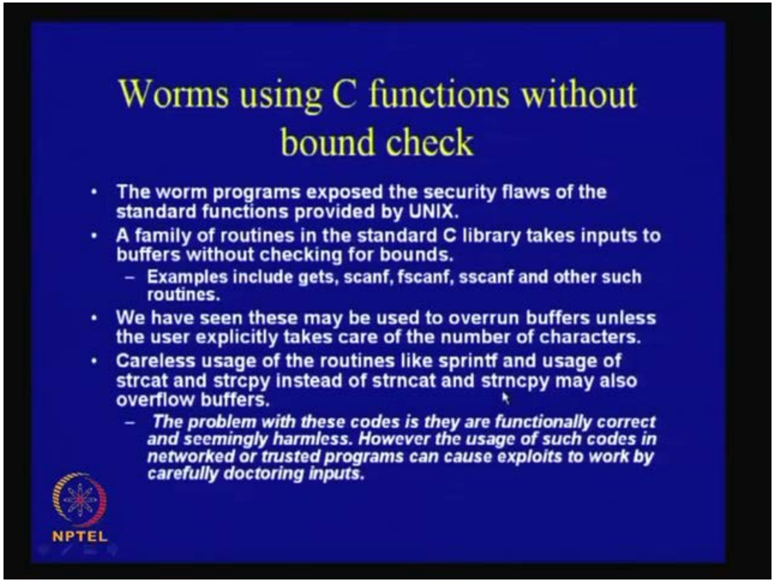


So, therefore, we will Study little bit about worms and viruses and understand their differences. So, now, worms are programs that can actually run independently, it can propagate a full working version of itself to other machines. On the other hand, viruses

are actually programs which cannot run independently. So, it is something like which are analogous to biological viruses.

So, they need to infect a host program and then, so that it runs and activates. So, therefore, the essential difference between a virus and worm is the infection step, like virus has to infect, virus cannot run independently; it has to infect and then only it can figure and work.

(Refer Slide Time: 35:02)

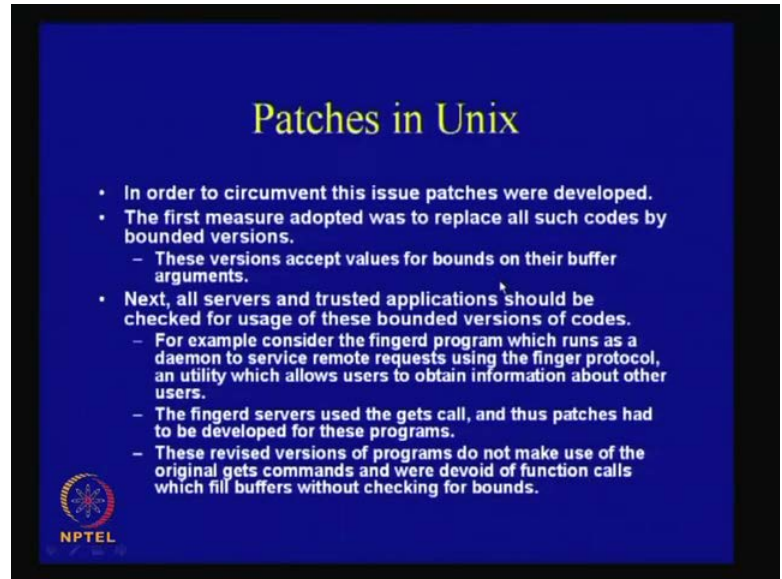


So, therefore this is the essential difference between worms and viruses. So, talking about worms first, so we have seen like I mean there are ample examples in literature, where c functions without checks have been used to trigger worms. Now, the worm programs exposed in various cases, the security flaws of standard functions which are provided by UNIX. A family of routines in the standard c library takes inputs to buffers without checking for bounds as we have just discussed, that is we can use like thing like gets, scan f, fscan f, sscan f and other such routines.

So, we have seen that these may be used to overrun buffers, unless the user explicitly takes care of the number of characters. So, careless usage of the routines like sprint f and usage of str cat and str cpy instead of their bounded versions may actually cause overflows. So, therefore, the problem with these codes is that, they are functionally correct and seemingly harmless; however, the usage of such codes in networked or

trusted programs can cause exploits to work by carefully doctoring the inputs. So, therefore, it is advice again that for security purposes, we actually replace this programs or this function calls with function calls like this str and cpy and str and cat, which are actually bounded versions.

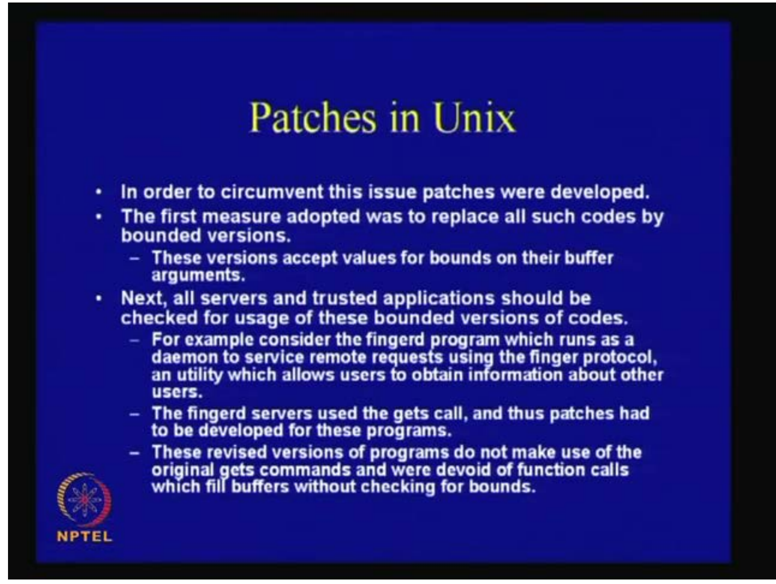
(Refer Slide Time: 36:24)



Now, that is a reason why we have actually patches in UNIX like, in order to circumvent this issues, patches are developed, the first measure which was adopted as actually to replace all such codes by the bounded versions. So, these versions accept values for bounds on their buffer arguments ok.

So, therefore, what is important is that, you may have an operating system and depending upon the security vulnerable vulnerabilities which exist, several patches have been developed, but if you do not update your patches in the operating system, then your system remains vulnerable. So, therefore, it is important for security purpose to update our operating system patches. So, that the plugs are sealed, I mean the security vulnerabilities are really plugged.

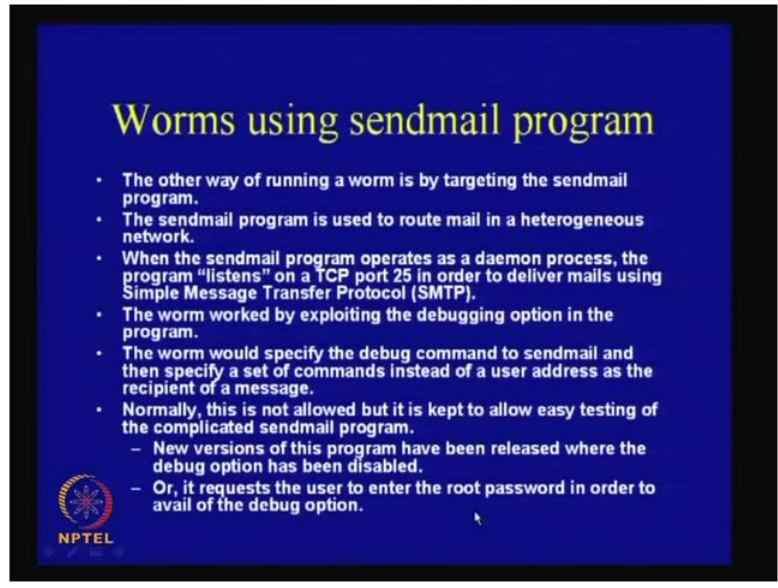
(Refer Slide Time: 37:13)



So, next all the servers and the applications should be checked for usage of these bounded versions of codes, for example, considered like this a typical example like of the finger d program. So is the daemon, right which we run as a daemon to service to remote requests using the finger protocol.

It is utility which allows users to obtain information about whoever are logged into the system, right who are working in the system. Now, the finger d servers use actually the gets call and thus, we people have to develop patches, to develop i mean rather replace this gets calls by their unbounded versions. So, therefore, these revised versions of programs do not make use of the original gets commands and were devoid a function calls which fill buffers without checking for bounds, ok.

(Refer Slide Time: 37:59)



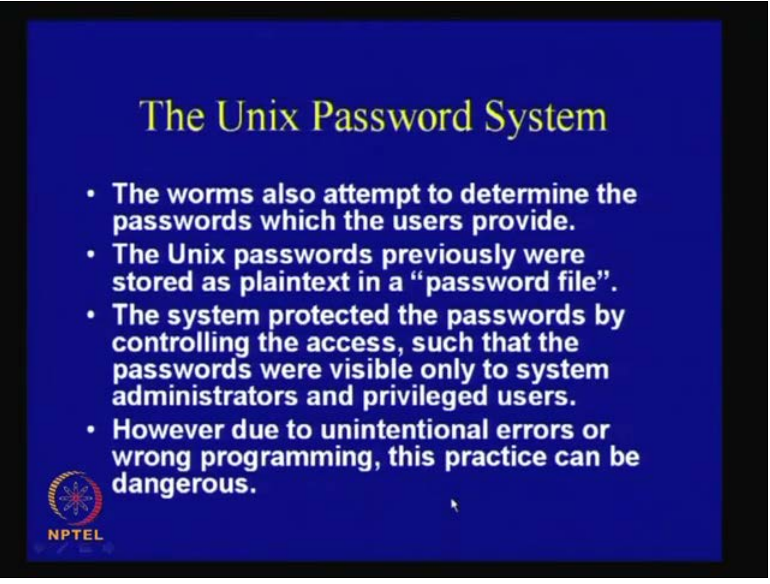
So, therefore, this is something which is actually done to circumvent not the function. So, that is no problem with the original finger d protocol or the finger d utility, but in order to make it. So, that ensure that does not leak it is not vulnerable against attacks like buffer overflows, we will actually replace them by their bounded versions and develop the patches right.

So, the other example which we can find out is the either segment program. So, what is the segment program? The other way of running a firm is by targeting the segment program, the segment program is actually use to the segment program is actually use to run route mails in heterogeneous network. So, therefore, if you have a heterogeneous network, then it is use to route the mail in a heterogeneous network. Now, when the send mail program operates as a daemon program, the program listens on a TCP port or print which is commonly the port 25, in order to deliver mails using the simple SMTP protocol, Simple Message Transfer Protocol.

So, by what by worm actually, when one of the worms which you have found that, was worked by exploiting the debugging option, this was there in the program. The worm would specify the debug command to send mail and then specify a set of commands instead of a user address as a recipient of a message.

So, normally this is not allowed, but it was kept to allow easy testing of the complicated send mail program. So, however, name this vulnerabilities was rejected, the new versions of this program have to be developed or released, when the, where the debug option was actually disabled. So, if you requests the user to enter the root or rather it may happen that, you either disable the debug facility or if you want the debug facility, then you have to ensure that, you have the you have the root. That is it has to be kind of kept privileged it is not a normal utility, but a privilege utility ok.

(Refer Slide Time: 39:54)



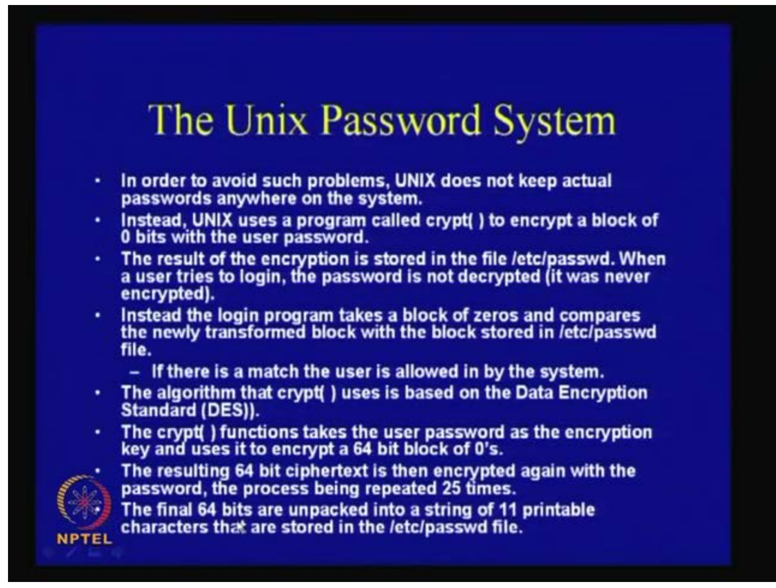
So, therefore, in context to this, we can actually one thing which is very important to discuss is the Unix password system. So, the worms several worms actually attempted to or other attempts to determine the passwords which the users provide.

Now, the Unix passwords previously were actually stored as plaintext in a password file. Now, the system protected the passwords by actually ensuring access controls, like everybody cannot use it, but; however, due to unintentional errors or wrong programming this practice can be dangerous. So, there was one example there actually there were two system administrators who were working on the Unix passwords or other they actually working together.

So, while one was working on editing the password system or password file, the other the other system administrator was actually creating banners for the day. So, banners means, whenever you log into the system you see those (( )) right.

So, therefore, what happen intentional or unintentionally is that, both the files got swapped. So, therefore, it was case like whenever anybody was using they were actually able to see all the files. This is the typical example where accidentally passwords keeping passwords in plaintext can be very unsafe.

(Refer Slide Time: 40:12)

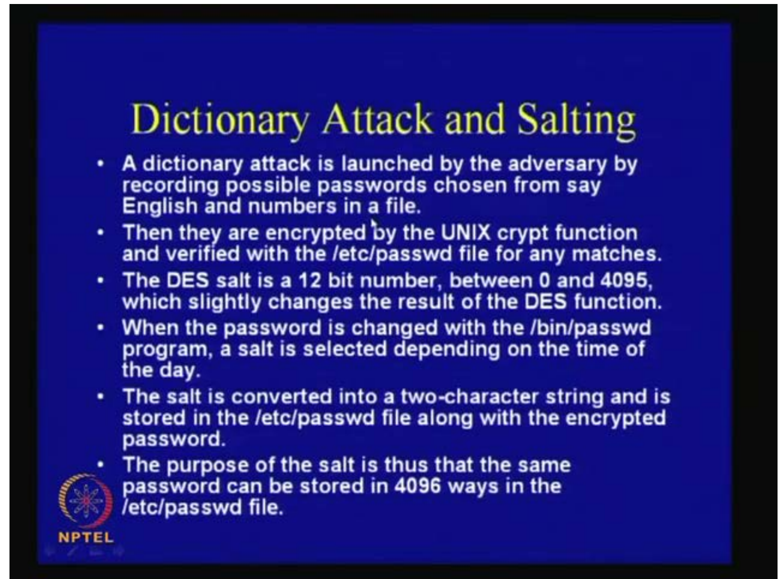


So, therefore, the present day Unix password system is to avoid keeping in plaintext and actually to use encryption. So, as if utility Unix program called crypt, so what it does is that, it actually encrypts a block of 64 0's and uses the base algorithm or des algorithm and creates a the corresponding cipher for the password and stores in the file slash etc slash password, I mean p a s s w o r d, ok.

So, idea is that whenever you login into the system, then you actually give in your login and this login is actually used to encrypt a 0 block of 64 bits and next corresponding. So, remember that, your password or whatever you are encrypting is never decrypted actually. So, it is a kind of ash function actually, although we say encryption, but it is a kind of ash because you are using I mean your key or your login is actually is used as the key for the encryption.

So, the crypt function takes the user password as the encryption key and uses it to encrypt a 64 bit block of 0 s the resulting 64 bit cipher text is then encrypted again with the p or password the process being repeated around 25 times. So, the final 64 bits are unpacked into a string of 11 printable characters that are stored in this particular file.

(Refer Slide Time: 42:27)



Now, wherever you change your files, I mean, I mean I mean important he actually use this utility password p a s s w d to change the files. Now, a particular the principle which is known as salting is used.

So, the reason salting is used is to prevent attacks, which are known as dictionary attacks. Now, dictionary attack is actually launch based on the assumption that, when somebody rather users normal user actually develops or rather creates a login or the I mean, creates a corresponding password, the normally the password having an alphanumeric password and it has actually a meaning in a English language ok.

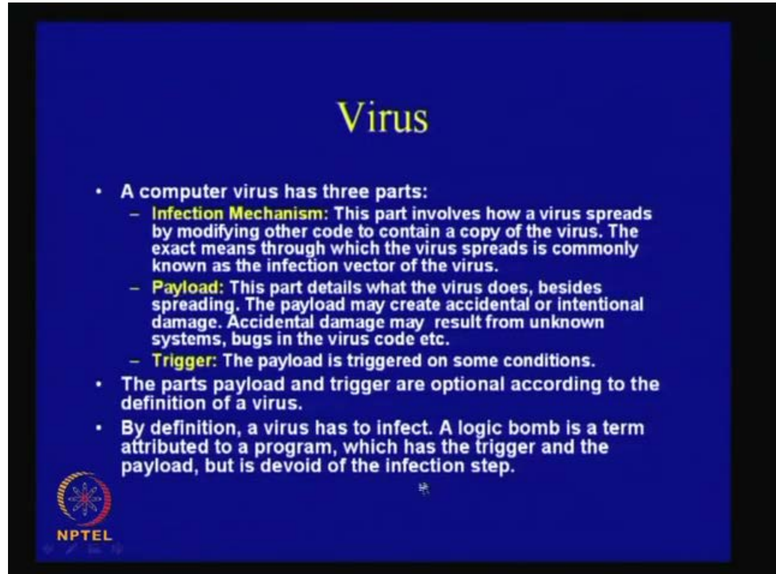
So, therefore, what may happen is that, if an attacker actually tries all the available entries in a normal dictionary and tries to see that which creates the match. So, this attack is commonly known as the dictionary attack. So, that actually reduces the entropy of all possible English language, I mean roman characters right. So, now, in order to prevent it, the people uses something which is known as a DES salt or a salt which is the 12 bit

number between 0 and 4095 which slightly changes that makes it complicated to do a dictionary attack, ok.

So, whenever the password is changed with this utility, a salt is selected depending on the time of the day, the salt is converting into a two character string and is stored in this file along with the encrypted password. Now, the purpose of the salt is thus, that the same password can be stored in 4096 ways in the slash etc slash password file. So, it just makes your job of doing a dictionary attack more difficult.

So, therefore, this is a this is a common way or other common principle of actually preventing against what is known as the dictionary attack which is launched by the adversary by recording possible passwords, which are chosen from say English and numbers in a files.

(Refer Slide Time: 44:22)

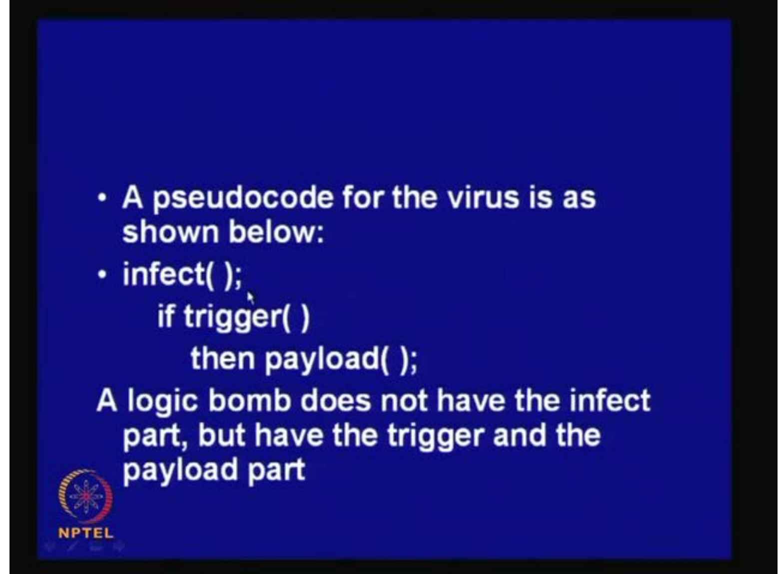


So, now, we come to that topic of on viruses. So, as I told you that virus have got three important parts, the infection mechanism, the payload and the trigger.

So, typically a pseudo code will look like this, that is that is an infection step, if there is a trigger then there is a payload. So, what the infection does is actually this is nothing but, how a virus copies into other codes, modify the code and contain a copy of the virus. So, this is sometimes called as the infection vector of the virus also.

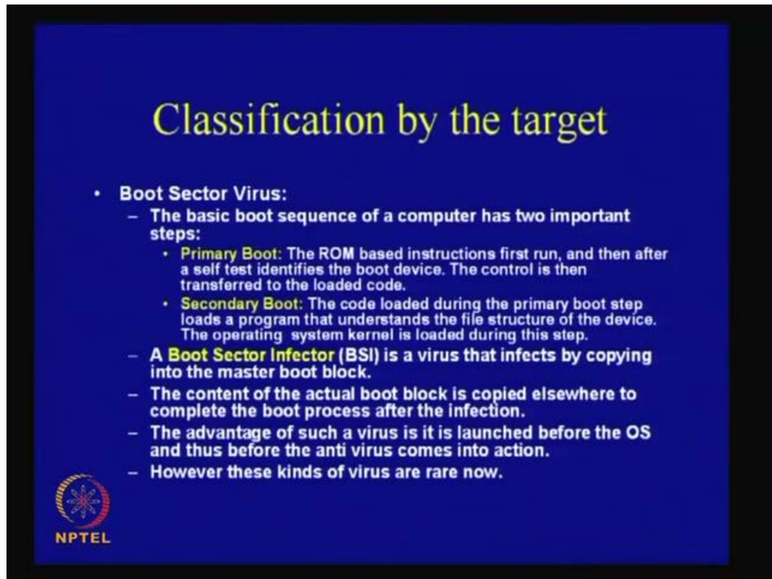
Then you have got a payload the part details what the virus does besides spreading like it has to do some malice. So, what is the malice specified? Now, the payload may also have some accidental to effects also, which may happen due to existing bugs in the virus code, it may do something unintentional damages also and there is a trigger. So, it is typically a brilliant condition based upon which a virus is triggered.

(Refer Slide Time: 45:19)



Now, there is something is called as a logic bomb. Now, the logic bomb is a typical case of a malicious program, where there is no infection part; there is only a trigger and there is only a payload. So, logic bomb could be like, whenever it could be it is also called like something which is called a time bomb that is depending upon the day of the time that comes into execution ok.

(Refer Slide Time: 45:40)



So, now, we will try to see what are the classifications existing for viruses. So, we will try to classify them based on the target. So, these are some very commonly known classifications. So, one is called a boot sector virus. So, it is actually exploits the basic boot sequence of a computer as we know that there are two important steps; one is the primary boot step and the other one is the secondary boot step. Now, in the primary boot step is the particular state when the system boots up is actually detects what is known as the boot device, right.

So, what this particular or other this category of boot sector infectors does is that, it actually hampers that master boot block. So, therefore, the master boot block gets co-opted and whenever the system boots up, instead of actually booting up from the master boot block, it actually gets affected by the virus. Now, one of the advantage is which exists on in the in this kind of virus is that, till now the operating system is not active, right.

So, therefore, the anti viruses are not active. So, therefore, the boot sector viruses are actually come before the antivirus actually comes into play, but however, these days this kind of virus are very uncommon, because it normally do not boot some things like floppies and other things. So, this is again, but still a primitive kind of virus, but an important class of viruses.

Now, there are some other viruses also which is known as file infectors, now these viruses typically target those files which is the OS considers to be executables. So, these viruses are generally something which is called as a pretending virus or appending virus. So, they are something like generally before the virus starts or it is generally after the virus starts.

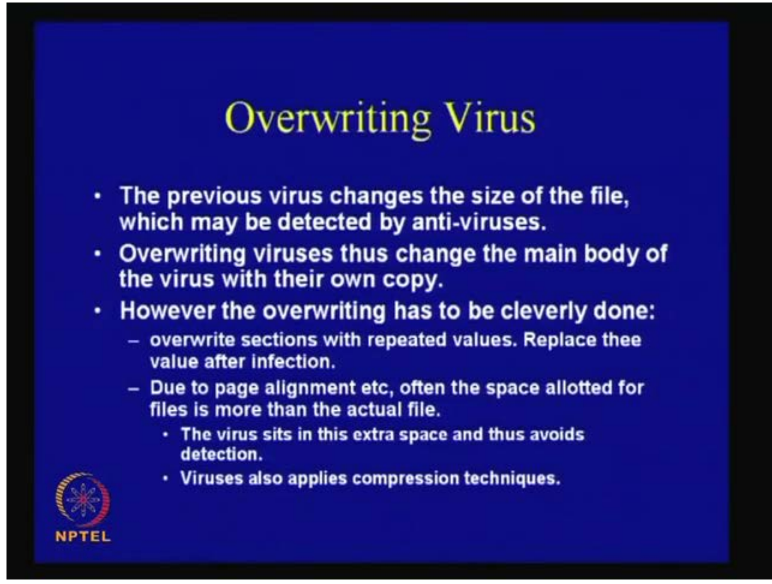
(Refer Slide Time: 46:57)

Executable File Infectors

- These viruses target those files which the OS considers to be executables.
- These viruses are generally either prepending (where the virus body is at the beginning) or appending (the virus body is at the end).

So, what may happen is that, So, therefore, what may happen is that typically you take any executable where this kind of viruses does is that, it either operates at a beginning of the virus or at the end of the virus. But, obviously you see that, there is a problem which comes with these viruses is that, the size of the files gets increased right. So, therefore, they can be detected by anti viruses.

(Refer Slide Time: 47:47)

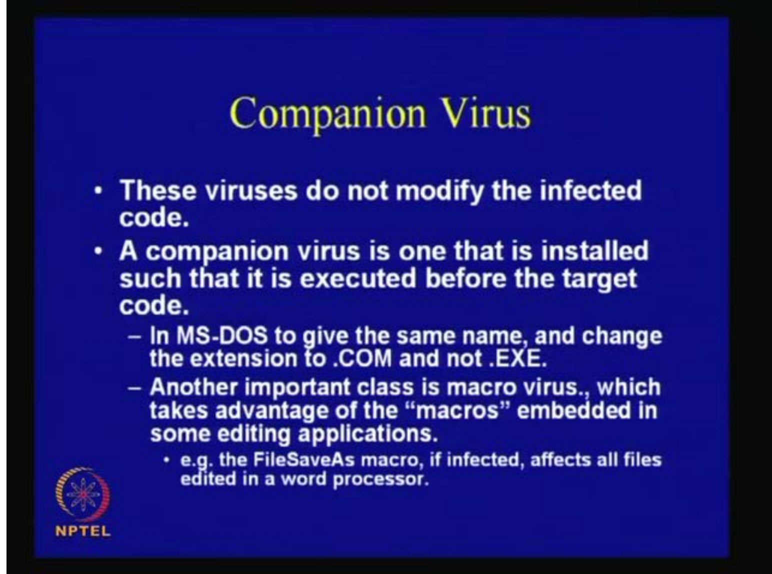


So, in order to prevent them, we have something which is which is A different class of viruses which is called the overwriting viruses. So, the overwriting viruses actually tries to find out the repeated values, which are there in a particular file and tries to actually modify them by some virus values and whenever the process or the bombing is over, then they again replace that by the original takes ok.

So, by that antivirus does not catch and in order to prevent the increase in the space, what is done is that, it uses the thing like page align if a features like page alignment. In order to because of this page alignment, we know that often the space which is allocated for a particular file is more than what is actually needed.

Now, this extra space is kept for I mean is because of the page alignment, but this extra space is being exploited to plant in the viruses, so is actually exploited to plant in the viruses. So, therefore, these are.

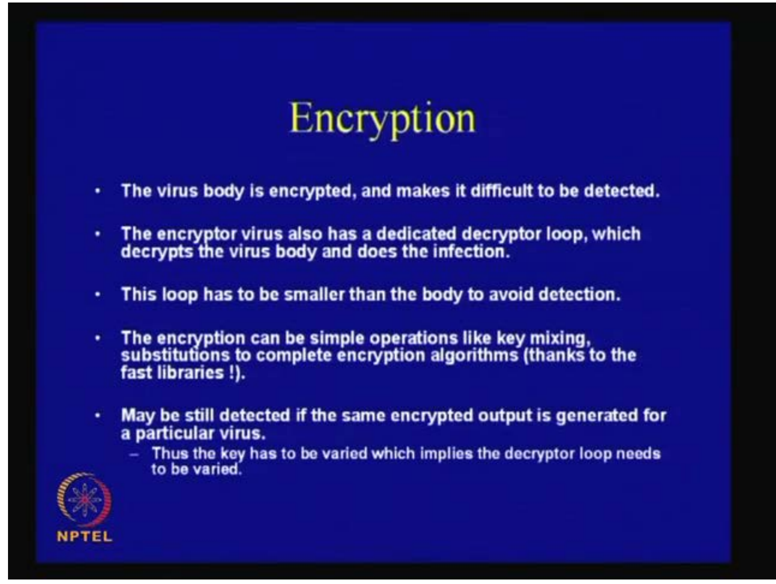
(Refer Slide Time: 48:40)



So, therefore, these are some common techniques and apart from them, you have also got something which known as companion viruses. So, these companion viruses are one which is installed such that it is executed before the target code.

So, they are I mean, what may happen is that, they may you may actually give the same name, but you can actually give a different extension which is actually executed before the original code. So, before the original code, the virus code is also is actually triggered and you can also have something which is called as macro viruses, where you are using may be a word thing, word editing file and you places may be a save as option and that save as option actually figures a macro which actually infects your so I mean, infects your file.

(Refer Slide Time: 49:24)



So, this is something which is known as companion viruses, when there are several viruses which actually uses encryption methods and encryption techniques. The encryption techniques are generally very light weighted, because they also has to be ensure that they are not detectable and are not understood by the corresponding anti-viruses.

So, based upon then you have got three kinds of viruses, one is called as oligomorphism, then metamorphism and polymorphisms. So, in this case you have got basically, you want to change the decryptor loops because wherever you have got encryption used for viruses, there has to be a decryptor loop right and what we would like is that we have a various decryptor loop, so that the antivirus does not catch them.

(Refer Slide Time: 49:40)

Oligomorphism

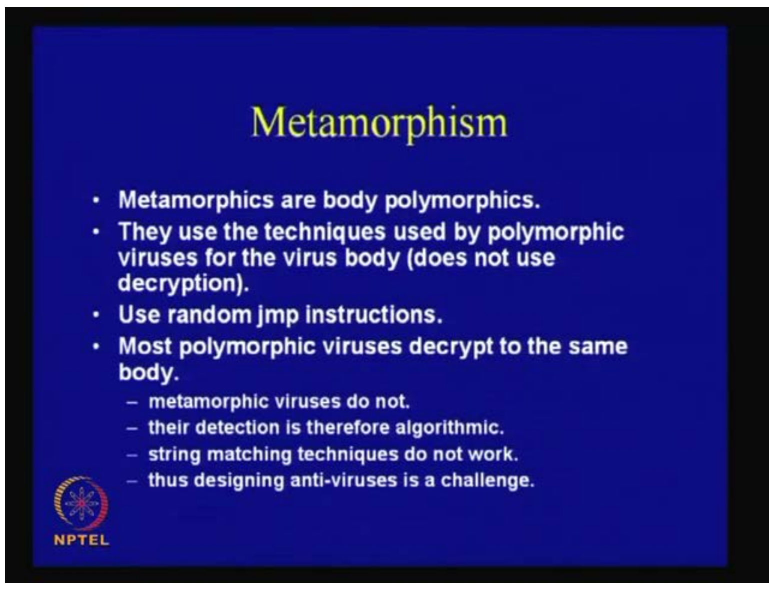
- The virus uses a varying key, thus the virus body gets changed with every encryption.
  - however the decryptor loop does not change for most of the part.
  - this gives anti-viruses to detect the virus.
- Thus instead having one decryptor loop have several.
- But the increase in difficulty is only marginal, as the number of decryptors increase to say 96 (Memorial Virus).

NPTEL

So, but in oligomorphisms the thing is that, the number of possible decryptor loops is limited like the memorial virus has it may be 96 decryptor loops. So, very small, but in as oppose to this in polymorphic viruses, there is an extremely large number of decryptor loops.

May be like 6 billion in case of some viruses and the virus actually changes the decryptor loops by using various mutation engines like, the techniques like which compiler uses like instruction equivalence, sequence equivalence, like parallelisms concurrencies. So, we can have AOS techniques like signal and weight kind of things which can be exploited so that the decryptor loop changes and it cannot be regretted by the anti viruses.

(Refer Slide Time: 50:48)



Now, you also have got something like metamorphisms, which is actually there like polymorphics, but only the actual body is actually encrypted, not the decrypt value, but the actual body of the virus. So, in these kinds of viruses, we use randomly jump instructions and most polymorphic viruses decrypt to a same body and so, you will see that in polymorphic viruses, after you do the decryption, there is a particular times which snap shot the time when the original body of the virus is there ok.

So, therefore, there you can actually have things like string matching's, which you can use for detecting where polymorphic viruses, but that you cannot do for metamorphic viruses, because metamorphic viruses are essentially there is no particular state, where essential the entire virus is disclosed. And therefore, the antivirus techniques for metamorphic viruses are actually quite challenge ok.

(Refer Slide Time: 51:42)

Points to Ponder

- How do modern compilers protect against stack overflow?
- Assume that the virus body has signatures belonging to the set: '01','010','101','1110','0101','11111','01010' and '11001'. Write a C program to parse the file and print an alert if a signature is found.

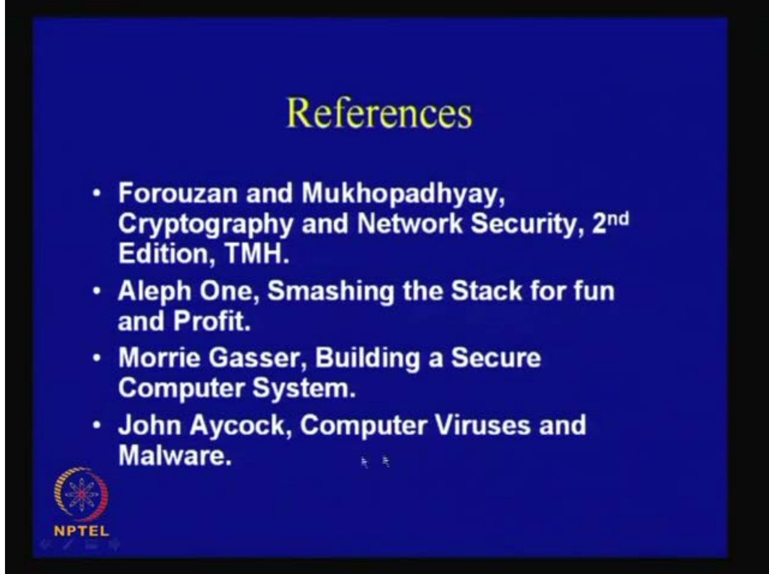
[Hint: Visualize the detector as a finite state machine]

NPTEL

So, end up again with thought on compiler or question like how do modern compilers protect against stack overflow? You can just do a little bit of survey on this and the other question is like assume that the virus body has got signatures belonging to the set like mention here like 01010 and so on and so forth till 11001.

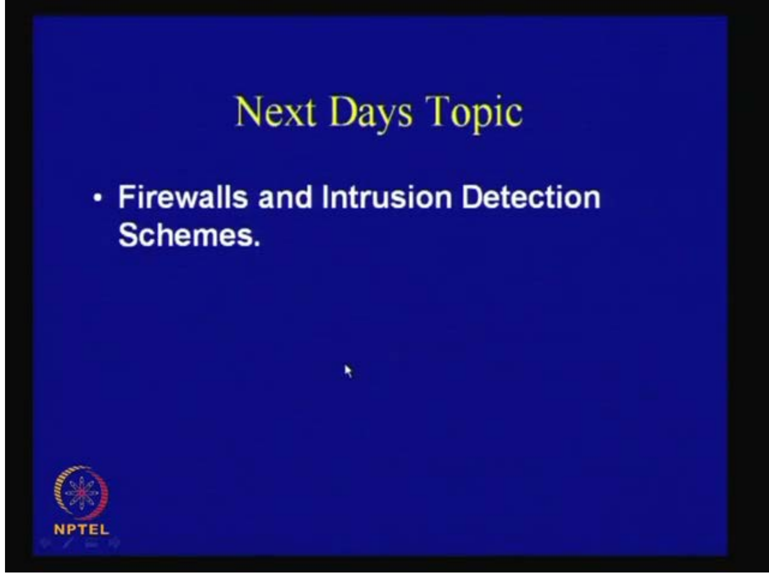
So, you can think of writing a c program to parse the file and print an alert if a signature is found. So, therefore, you can visualize, rather implement or visualize the detector as a finite state machine. It is one class of anti viruses which are actually builds up like finite state machines, which you actually detects for or looks for this signature in the virus files.

(Refer Slide Time: 52:30)



So people used data structures like try to detect them actually, so there is some references again, this is a very interesting reference smash mean smashing the stack for fun and profit by l f 1. And you can use this building a secure computer system by morrie gasser and this book on computer viruses and malware, where you can find an excellent depositories and discussions on viruses and malwares.

(Refer Slide Time: 52:53)



So, in the next day, we shall discuss about firewalls and intrusion detection schemes. So, and that could be the next thing when we see about, how to tackle security issues in networks, thank you.