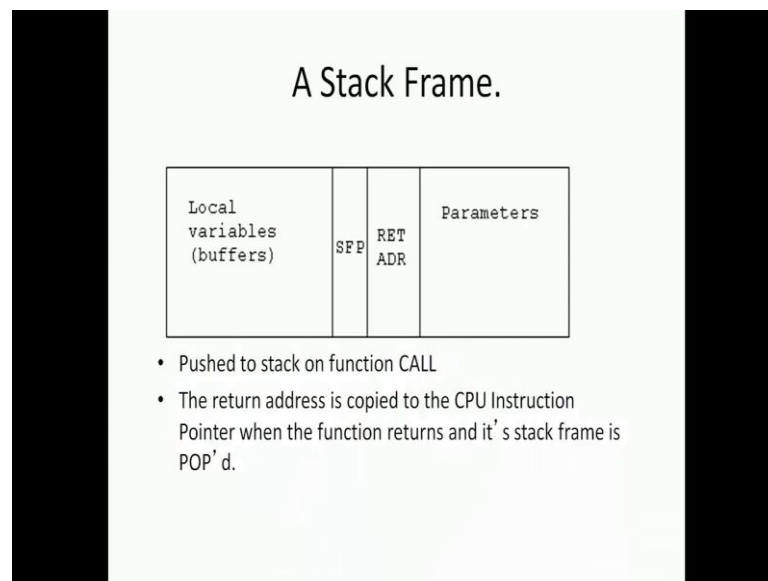


Information Security – II
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 05
Architectural Aid to Secure Systems Engineering
Session – 4: Stack Smashing

It till session 3, we looked at Stack Smashing sorry, we looked at Functions and Calls and how stacks use function calls, stacks are used for function calls. Now, what we will be demonstrating in session 4 is, how is this function call and stack together is going to cause a Vulnerability? And that vulnerability we term it as stacks smashing which was will studied in the literature.

(Refer Slide Time: 00:48)



Let us look at what is stacks smashing. So just to recap, what happened when a function was called a stack frame was created, in that stack frame there was all the local variables. So, as you see in the slide these are basically buffers were the local variables are stored then there was a stack frame pointer, there was a return address and the parameters that are to be used by the calling for called function. When a calling function calls the next function, first there will be parameters that are past that will be stored then the return

address will be stored then there will be a stack frame pointer and then there will be a buffer created for the local variables of the called function right. So, what you see on the right most side are the parameters which are, this is the stack frame of a given function. When a function is called whatever, you see on the right hand side from the right hand side the parameters or the parameters for the function then the return address that is after execution of the function where should I go then there is a stack frame pointer and then there are some local variables that are stored here right.

So, this is what would be a stack frame for a given function. What happens after the function is finished, all your variables are popped or you go to the return address that is there in the stack and then just do a return and it will back to the original function which was calling it right.

(Refer Slide Time: 02:32)

The buffer overflow vulnerability.

Local variables (buffers)	SFP	RET ADR	Parameters
Buffer: ZZZZZZZZZZZZZZZZZ			

- The user injected data writes beyond the unchecked buffer length, overwriting the stack frame return address!

Now, what is this buffer overflow? So, lot of see you create something called a buffer. A character care star yes is a buffer, where you can store lot of characters. Now, when I create a buffer where I can store lot of data elements, I put a limit for that buffer right, so I say 100, I can store up to 100.

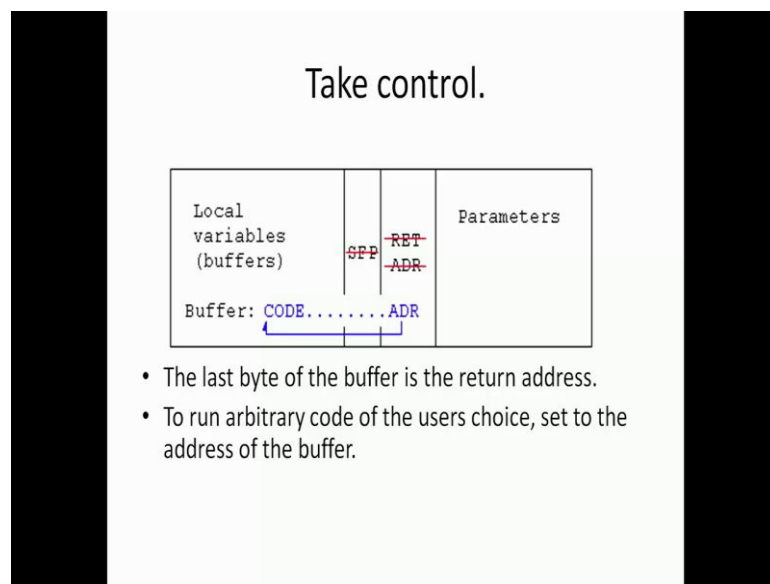
Suppose, I cross that 100 value sometimes you check, sometimes you do not even check.

If I want to check whether for every access, suppose I give a character array of size 256 I can store up to 256 characters there, but if I am going to violate this 256 and go for the 257 somebody has to check whether I am violating. And every time if I say every time we are trying to access that array a check for the bounce, what will happen? The performance will take a major hit. Every time I am going to read something from that array first I will check the bounce and then go and access that array, then the performance is going to take a major hit. In many of the compiled code, this is compile has become important they do not do this check. So, when somebody starts writing into the buffer they will keep on writing and sometimes they may even over shoot it and write into the next variable. In the memory you have defined a buffer after 256 that will be 256 bytes say it is a character array there will be 256 bytes, then there will be something more on top and below. So, as and when you start filling up the array and you breach the bound then you go and write into the next variable right. That is why sometimes and some of the old see programs if you see when I declare I have a character array and then, some other integer variable following it and so if you go and write something into the character array and if it is go about the bound the integer variables value will be change. So, these are something's which you could have seen in some old compilers where strong type of bound checking's are not there, ok.

Now, this vulnerability is basically exploited. So, what are we exploiting now? I have a stack which is provided by the architecture, I have a character array that is provided by the programming language and that is the being compiled and so there is a programming language, there is a compiler, there is an architecture and these 3 together is as created this infrastructure of doing function calls correct. Do you follow? Now what I do here I am exploiting that particular vulnerability where, the compiler does not check the bound, where the compiler does not produce an assemble code which will go and check the bounce. What happens; when, the actual program is executing I can keep writing into that buffer infinitely. So, what you see on the screen is that. The buffer is a part of the local variable of a function and what I do I keep on writing, so it breaches it goes on writes into the stack frame pointer then it goes and even corrupts the return (Refer Time: 06:06) Since the stack is accessible to the called function also, as a called function if I have a buffer I keep writing into it and nobody checks it. You got this. So, by this what I have done I have corrupted my written address I have gone into the parameters of the

other fellow also, so I could corrupt my stack right. This you may made it unintentional because your program we just trying to read a character array and unfortunately the string is more the 256 bites, that is an unintentional way of doing it. And what would happen if it is unintentional? Then you have a some written address which will not be part of you will go to some junk and then the will program crush and that is why some unintentional bugs actual create the crush. But, if I know that I could breach this buffer can I do something intelligent here and that is what we call as stack smashing.

(Refer Slide Time: 07:14)



So, what we do there? What I do is I write a value into that buffer which is actual an assembly program and I write, so since I can go beyond that buffer in the written address part as you see I write an address that will point to the start of this code. What I have written into the stack is an assemble code and in that part where there is a written address I will change that current address to point to the start of this code which is inside this buffer. Now what happens when I execute a written, which written address will come? This new written address will come and what will happen it will start executing the current code which I have written. So, I am a calling function, I am called function, I have access to that local variables, I keep filling up that buffer with an assemble code of that machine and I go to the top and I have some written address in that written address I go and change that address to point to the start of my assemble code which is inside the

buffer itself. Now I execute a written, what will happen? It will now start executing the assemble code which I have put and in whose privilege, when I do a written the privilege level will shift to the called calling functions. If that calling function is a supervisor function and I am a user function, now I start executing some code my code with a super user privilege, are you getting this I start executing my code with a super user privilege.

(Refer Slide Time: 09:11)

Stack Buffers

- Suppose Web server contains this function

```
void func(char *str) {  
    char buf[126];  
    strcpy(buf, str);  
}
```

Annotations for the code:

- Allocate local buffer (126 bytes reserved on stack) - points to `char buf[126];`
- Copy argument into local buffer - points to `strcpy(buf, str);`

- When this function is invoked, a new frame with local variables is pushed onto the stack

The diagram illustrates the stack structure. It shows a horizontal bar representing the stack, with an arrow pointing left labeled "Stack grows this way". The stack is divided into several sections from right to left: "Frame of the calling function", "Top of stack", "str" (Arguments), "ret addr" (Execute code at this address after func() finishes), "sfp" (Pointer to previous frame), and "buf" (Local variables).

This particular slide basically describes, what I have been talking, suppose I have a web server and it has this function `char *str` and I have 126 bytes that are allocated. Now I am copying some string into this buffer, whatever that function `strcpy` that `str` which is an input parameter that will copy into this buffer right. Which is the calling function? The calling function is this function that is calling that thing calls `strcpy`. Now `strcpy` does not check the bounds, it does not check whether the length of the string `str` is less than or equal to 126 it does not count that, it does not check it. What happens now this function comes up, when this function is invoked actually a new frame is actually created with local variables in the stack, and what I do?

(Refer Slide Time: 10:12)

What If Buffer is Overstuffed?

- Memory pointed to by str is copied onto stack...

```
void func(char *str) {  
    char buf[126];  
    strcpy(buf, str);  
}
```

strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations

This will be interpreted as return address!

Instead of into the buffer I write this str some 126 bites plus it will write something more. And what will be there inside that string, it will be an assemble program and that program the end of that program will have an address which will point to the start of this code.

(Refer Slide Time: 10:28)

Executing Attack Code

- Suppose buffer contains attacker-created string
 - For example, *str contains a string received from the network as input to some network service daemon

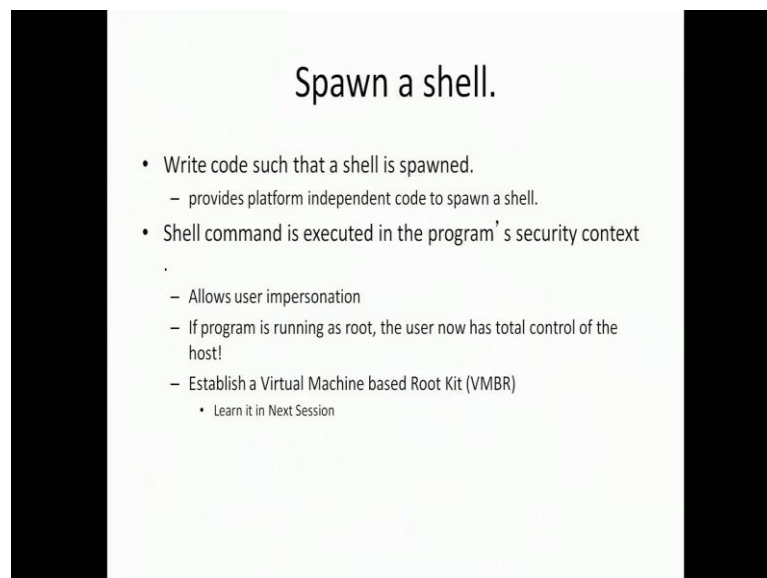
Attacker puts actual assembly instructions into his input string, e.g., binary code of `execve("/bin/sh")`

In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address

- When function exits, code in the buffer will be executed, giving attacker a shell
 - **Root shell** if the victim program is setuid root

So, my entire assemble program along with an address which is pointing to the start of this code is available and this I write it into the stack and I now execute a written. And what happens now? My code start executing and that codes starts executing with the super user's privilege. What will happen if suppose a super user is a root and this will be a very small code 120 bites, what code can I do? I can spawn a shell, you know what a shell is in your g o i you can create no terminal right that is called a Shell, bash shell seashell these are all available. I can create a shell. Once I create a shell with root permission, what can not I do? Everything I can do, I can go and delete account create account you go to any user file everything. This was one of the primitive attacks. In modern things there are lot of face by which could avoid this thing, now there are solutions for this problem. But why I am talking about this problem is to basically tell you a good case study, with in a very small you know a small period of time. I have to give you a case study were there was a compiler involved, there is an application program involved, there is an architecture involved all 3 things together has actual created a vulnerable scenario. So, I thing this is 1 very good motivating case study to understand how this interface can create problems ok.

(Refer Slide Time: 12:14)



Spawn a shell.

- Write code such that a shell is spawned.
 - provides platform independent code to spawn a shell.
- Shell command is executed in the program's security context
 -
 - Allows user impersonation
 - If program is running as root, the user now has total control of the host!
 - Establish a Virtual Machine based Root Kit (VMBR)
 - Learn it in Next Session

What will happen if I spawn a shell? I have a shell with through privilege I can do lot of thing and the thing is that once there is a facility to start executing program with through

privilege then I can be passive, I can be active, passive means I will just watch what you are typing right, so I will just keep logging your keyboard entries something called a key logger. Then I can all your password everything where you are accessing I can monitor, so whatever you are typing all the user inputs if I am able to gather then so many things I could gather or I can go and kill your program, tomorrow is a conference deadline you are my enemy all the pdf file that are there will be arrest every 15 minutes gone right, so I can do this. These are some of the things and I can impersonate right I can go into your email account and send a mail to your guide saying you are an idiot, so these are all possible.

So, one of the things that due through stack smashing what they enable, is what you call as Virtual Monitor Based Root kit. Virtual machine base root kit which will be seeing in the next session, it is called VMBR. How it is going to act? And that is going to be the real way by which this facility of stack smashing has been utilize for this. So, to some up what we have understood in this session is before we go into VMBR is that the way the operating system, the compiler and the architecture interacts has to be understood very nicely, as to be understood with very good detail right? You cannot just have a very casual look into this right it has to be understood in very great detail. And if you have such an under standing then you will be in a position to appreciate the important aspects of security. So, security is not something that is explicitly seen, it is not isolated to one entity. I do an architectural analysis I can get lot more about security this is not going to be possible because security basically comes in the way by which multiple things in the system interacts, the vulnerability manifest itself because of intersections of multiple components in your system and it is not secluded to one percent. You are getting this?

That is why security becomes extremely complex. It is not that I have an heart attack I have sprain in the leg, so it is some multi organ failure which will kill your system right. You understand? And that is were know the complication comes right. This is this stack smashing though it is a very old concept and there is lot of contra measures for this, but I prefer to teach this in this course so that you have an understanding of how multiple things can work together.

So, before we go to the next session I will open up for questions in this session.