Information Security – II

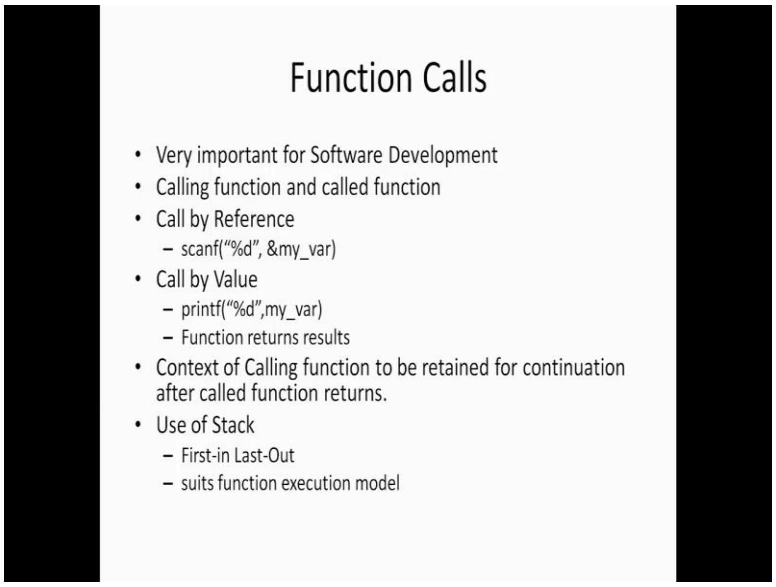
Prof. V. Kamakoti
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 04
Architectural Aid to Secure Systems Engineering
Session 3: Function Calls and Stacks

In the previous session, we stopped off with saying that there is something called a buffer overflow which I want to basically start talking because this buffer overflow that I am going to cover in this particular session, next two sessions is an example of vulnerability in which there is a participation from the compiler, there is a participation from the operating system and there is also a participation from the architecture. So, the architecture operating system and compiler collectively is responsible for vulnerability, and that is what I am going to present in the next two sessions. So, in this session as I told you, then assured you that this course is extremely self-sufficient and you need not have any free request of course some basic programming that I hope everybody knows.

I will now talk about Function Calls and Stack, right. It is a very preliminary subject, but to have a completeness and also revise if in case some of you have forgotten, I will now talk about Function Calls and Stack in this particular session 3.

(Refer Slide Time: 01:29)



Function Calls

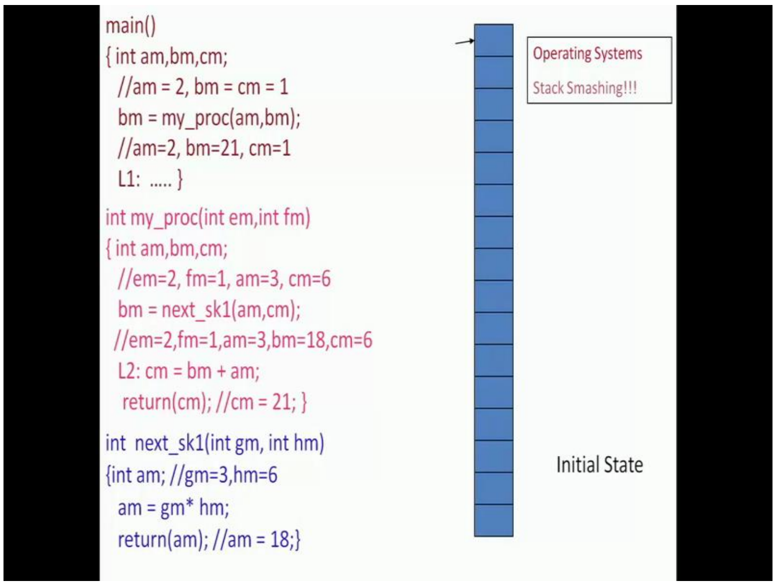
- Very important for Software Development
- Calling function and called function
- Call by Reference
  - scanf("%d", &my_var)
- Call by Value
  - printf("%d",my_var)
  - Function returns results
- Context of Calling function to be retained for continuation after called function returns.
- Use of Stack
  - First-in Last-Out
  - suits function execution model

So, what is function call? The function call is very important for software development. Why is function call very important for software development? It is because this gives you a modular approach to developing software, a software reuse. It is also possible because of function calls, correct. You write a function and that function can be reused by somebody else. That is how libraries have come up, our math dot h, your even stdio dot h, everything are, all libraries that have developed by someone which you are using. So, function calls are extremely important for software development. Then, there are two, and in a function call there are two fellows. One is a calling function; another is a called function and they are two-way by which you can call a function. So, between a calling function and the called function, there is some communication that needs to happen. The calling function actually sends certain parameter to the called function. Based on those parameters, the called function evaluate something and return backs a value and this return back can happen the way by which you pass the parameters. You can pass the parameter either by reference or by value.

A very simple example of passing by reference is your scanf, where you go and say go and store the value of your reading in a particular location. So, in this case my rar as you see is a variable, its integer variable. When I say amperes and my rar, it gives the address where that my rar is stored, right? So, I pass that address to the scanf. So, whatever I am reading there, that integer what I am reading there is gone and stored in this address. So, when I come back out of scanf, I go and access my rar location. I will get the value of the variable, you get this. When I am printing the value, I have to tell what to print. So, there is no need for an address. I just send the value and it goes and gets printed. So, printf percentage my rar actually takes the value of my rar and gets it printed. So, these are the two ways by which I could pass parameters between a calling function and a called function. So, the function actually returns; many of the functions return certain values. Even the printf actually returns a value scanf also returns a value. Do you know what value scanf returns? It is the number of number variables that are basically red.

Now, the most important thing is when I call a function, it does certain things. It starts executed. So, I am calling function. When a call function, it starts executing and then it gives back me the answer, right and when I get the answer, I should start executing form the point where I left, right. So, the most important thing is the context of the calling function has to be retained for continuation after the called function finishes, right. So, that is very important and that is where the programming languages have grown. They allow you, they give you an infrastructure by which you can call a function. The functional get executed and then, you return back and you start from the point where you left. So, there are certain variables that may get manipulated because of this function call, but all the remaining variables should have their own value and I should be in a position to start from the point where I left. Are you getting this and that is very important and to maintain this context, a stack, a data structure name stack is used. The stack flow a very simple policy call first-in, last-out policy. Whatever, I push first that will come out last. So, if I push a b c and what you call the other operation pop, I will get c b a in the opposite direction and this first-in, last-out model or last-in, first-out model would means the same suits the function call model as follows why it suits that.

(Refer Slide Time: 05:45)

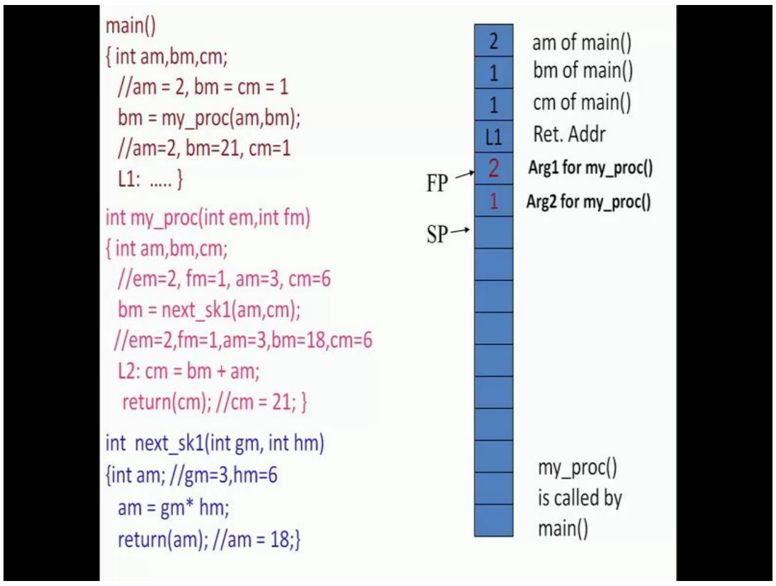


```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1:  ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

So, look at this slide. On the right hand side you see stack, the blue color. What you see is a stack. I hope all power points exists, all cameras show it is blue. So, the blue color you see is a stack and there is one small arrow on the top, it is called the top of the stack. This is where you start pushing in something into the stack. So, the initial stack, state of the stack is that nothing is there in the stack. Now, I am going to execute the program that you see on the left, right. So, there is a main routine, there is a main function and it has three variables am, bm and cm and then, it calls a function call my proc and it passes two parameters into it, namely am and bm that my proc execute something and it will return an integer back which is stored back in bm inside my proc. Again it has two input parameters em and fm what you see in the red color there and then, it has three integer variables am, bm and cm. Inside that it calls another function what we call it as next sk1. It passes two parameters next sk1. Actually again has two input parameters and some variables. It does some computation and it returns a value. So, main calls my proc, my proc calls next sk1, next sk1 executes it, returns back. Once it returns back a value, it returns back to whom? It returns back to my proc. That value return by next sk1 is assigned to bm and then, your my proc starts executing.

Please see the red. There my proc starts executing from l 2. It does some computation and it returns a value that return value goes to the main routine which is assigned to bm again and now, the main routine starts executing from l. So, are you getting the context of this? So, what happened is, main called my proc, my proc called next sk1, right. So, who was first fellow main went in starts executing first, then my proc starts executing next, then next sk1 starts executing last who finished first. Next, sk1 finished first, then my proc finished next, then main started executing it will finish last. So, that is why the fellow who starts executing first finishes the last in the contexts of functions, and that is why we call it as a last-in first-out policy. Since it is last-in first-out, I can use a stack basically to execute this function, ok.

(Refer Slide Time: 08:55)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

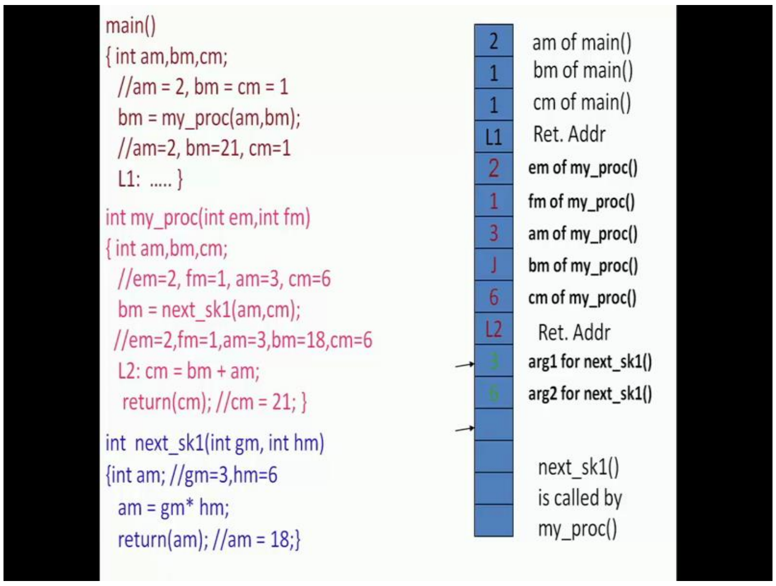
	2	am of main()
	1	bm of main()
	1	cm of main()
	L1	Ret. Addr
FP →	2	**Arg1 for my_proc()**
	1	**Arg2 for my_proc()**
SP →		

my_proc() is called by main()

Now, what happens here when I start executing this? Please note that when main starts executing, let us say before it calls my proc before main calls, my proc the value of am, bm and cm are 2 1 and 1. So, in the stack on your right hand side, there are locations reserved for am, bm and cm of main and there you see 2, 1 and 1. Now the function call happens my proc is being called automatically. What happens is the hardware, the architecture loads the return address, namely l here because after my proc finished execution, your main should start receiving from l, right. So, that l that returned address is basically stored in your stack who does this. The compiler just compiles a instruction called call, the call when it is executing in the architecture. The architecture loads the return address. So, who is responsible for putting that l there? Architecture and then, the two parameters that are passed, namely am and bm, that is 2 comma 1 is also now loaded into the stack.

Now, what happens is my proc starts executing. So, where will my proc get the value of the parameter? It is already there in the stack. So, that is why there is something called fp which we call it as the frame pointer and there is sp which you call as the stack pointer, right. The frame pointer points to where the variables for the current routines which has come from the calling function are stored. So, fp points to 2 and next is 1. Now, the stack pointer is pointing to the next empty location there. Now, my proc starts executing. So, my proc gets the value for those two input parameters, the em and fm. You see on the red on your left hand side, the red path that values are 2 and 1.

(Refer Slide Time: 11:03)

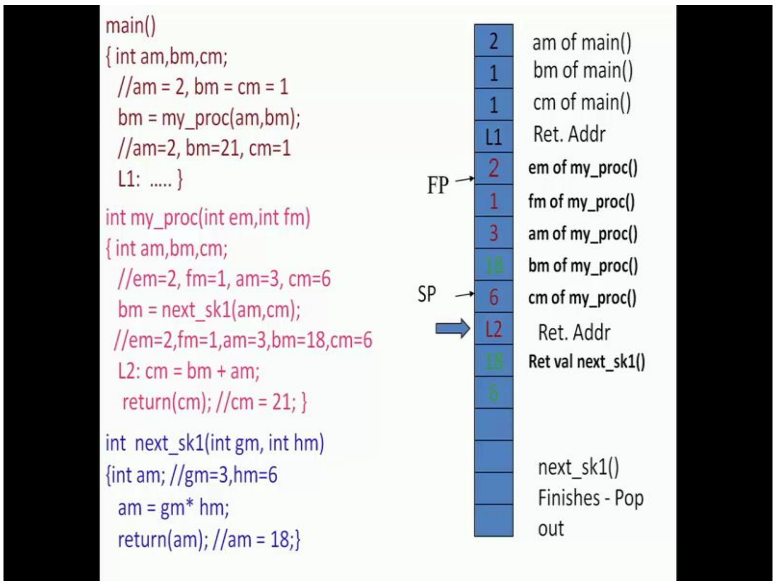


```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
  //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

2	am of main()
1	bm of main()
1	cm of main()
L1	Ret. Addr
2	em of my_proc()
1	fm of my_proc()
3	am of my_proc()
J	bm of my_proc()
6	cm of my_proc()
L2	Ret. Addr
3	arg1 for next_sk1()
6	arg2 for next_sk1()
	next_sk1()
	is called by
	my_proc()

Now, my proc starts executing now what happens there my proc has an a m it has a b m and it also has a c m. So, so a m takes a value three b m is not assigned it is having a value junk it can value j then c m as a value 6. So, before my proc reach reaches next sk1. So, these are all the values for all it is local variables. So, there are 3 local variables for my proc a m b m and c m and those are also getting loaded into the stack then Now, it passes to values 2. Now, it calls next sk1. The moment it calls next sk1, the compiler will compile it as a call, right. The assembly language instructions called call, that call will execute and your architecture will now load. What the l, 2 the return address that is after my next sk1 completes, where should I start executing that? L 2 is loaded into the stack, the n 3 and 6, the parameters that are passed for next sk1 are also loaded into the stack, right. Now, next sk1 executes. Now, next sk1 takes 3 and 6 and it will return the value 18, ok.

(Refer Slide Time: 12:27)



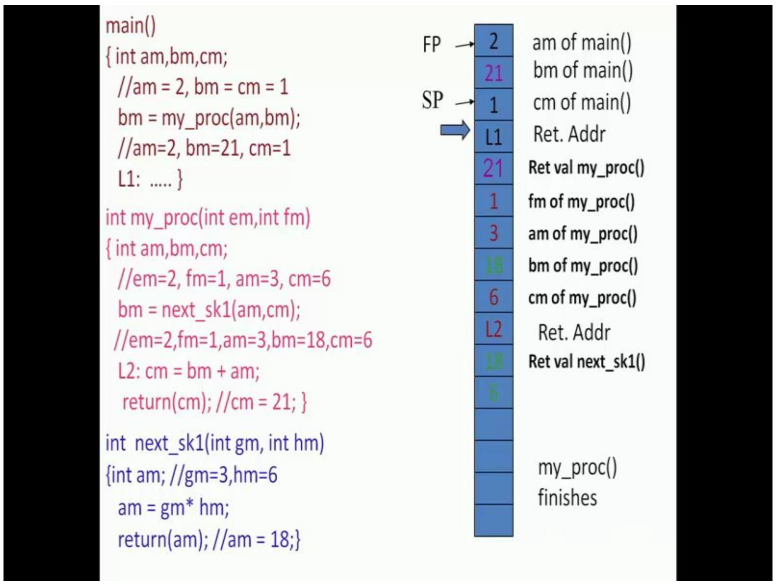
```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
  //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

	2	am of main()
	1	bm of main()
	1	cm of main()
	L1	Ret. Addr
FP →	2	em of my_proc()
	1	fm of my_proc()
	3	am of my_proc()
	18	bm of my_proc()
SP →	6	cm of my_proc()
⇨	L2	Ret. Addr
	18	Ret val next_sk1()
	6	

next_sk1() Finishes - Pop out

So, it returns the value 18. Where it will return the value? It will return the value back to the same stack. Now, when you return, now what happen is the control goes back to l, 2 and where you get the value of l, 2, it is already there in the stack. So, I take the value of l, 2 from there and before starting executing, I have to make bm as 18. Note that in the stack bm actually become 18 which is the return value from next sk1. Now, the program will start, the next proc will know my proc will start executing from l, 2 right? Now, what happens here? So, bm is 18 am is 3. So, your cm now should become 21, right?

(Refer Slide Time: 13:14)



```
main()
{ int am,bm,cm;
  //am = 2, bm = cm = 1
  bm = my_proc(am,bm);
  //am=2, bm=21, cm=1
  L1: ..... }
int my_proc(int em,int fm)
{ int am,bm,cm;
  //em=2, fm=1, am=3, cm=6
  bm = next_sk1(am,cm);
 //em=2,fm=1,am=3,bm=18,cm=6
  L2: cm = bm + am;
   return(cm); //cm = 21; }
int  next_sk1(int gm, int hm)
{int am; //gm=3,hm=6
  am = gm* hm;
  return(am); //am = 18;}
```

So, there answer is again stored back into the stack by my proc. So, if you see on the stack on your right hand side, you have 21. Now, what happens here is, now your programs starts executing from l, right. Your Ret address is that. So, what has happened now? So, when we just quickly see this before we wind up this session, what has happened now is as follows at every point when I call a new function, the moment a function is called, there is a frame created for this. This is called a stack frame, right. In that frame all the variables, all the local variables are basically given some place where, you can go and store the values of this local variables you are getting. As I go to the next function, these local variables are not touched. The next function works on its own stack frame and after it finishes, it just passes the value back to the previous stack frame.

So, if I quickly go back, the stack frame creates that for main and when my proc executed, there is a new stack frame created for my proc. This my proc which is not touching the stack frame of the main. So, all those variables there remain untouched. So, when my proc completes and goes back to main except for the variable that is going to be returned by my proc and that is going to be manipulated by my proc which is b here, all the other variables, the values are remaining same, right. I hope you understood this. This is how stacks and function calls work. Why should I need such an infrastructure? It is because see please note I have made one careful decision here. I have made one very interesting thing here that I have named all variables same in all the three functions. Most of the variables are same. So, am of main is different from am of my proc which is different from am of next sk1.

Why is this important? It is because otherwise if I say every function should have its own distinct name, then I cannot reuse. So, I say I will use your function only if you do not use the name am. So, everywhere you put some Aadhar id in your hopefully it is unique and write your variable names, right if we do that. We cannot do that. So, this gives you independence in some sense. It is isolating the execution environment of my proc from the execution environment of main, right. It isolates your next sk1 execution environment from my proc from main, right and when I keep calling functions, what is happening here is every new function is working in isolation without touching the other functions. This is how function calls are executed.

Now, one other thing before I wind up this session, I should basically tell you that I am using the same stack, right. Why I am using the same stack in this environment? I am using the same stack because I want that function to communicate with me and I am using the stack as a medium for the called function to communicate back to the calling function, and the calling function to communicate to the called function. How do they communicate? I send you variables and you send me the result. I am the calling function, you are the called function and I use a stack as a medium for this communication as you have seen here. So, with this small input, I wind up this session and we will go the next session.