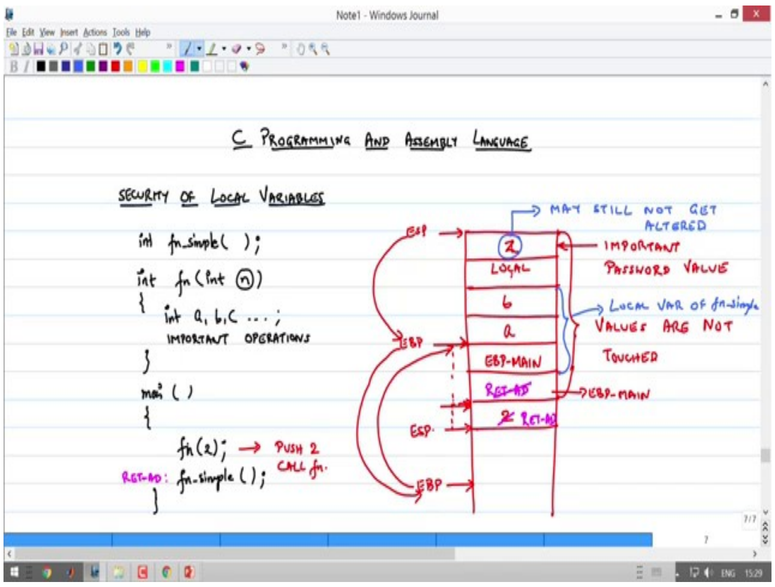
C Programming and Assembly language

Prof. Janakiraman Viraraghavan
Department of Electrical Engineering
Indian Institute of Technology, Madras

Lecture - 20

Welcome back to this course on C programming and Assembly language. So, we are in the last module and we are also in the final lecture of module 4. In the last couple of lectures, we discuss the impact of assembly translation on various aspects like variable argument list, performance impact on recursion, because of recursion and loops, then we also discussed about some performance improvement by speeding up certain functions like string compare, string length and so on. In this last lecture, I would like to discuss the concept of security of local variables.

(Refer Slide Time: 00:49)



C PROGRAMMING AND ASSEMBLY LANGUAGE

SECURITY OF LOCAL VARIABLES

```
int fn_simple( );

int fn(int n)
{
    int a, b, c ... ;
    IMPORTANT OPERATIONS
}
main ( )
{
      fn(2);          → PUSH 2
                        CALL fn.
RET-AD: fn-simple ( );
}
```

So, what I mean here is that a local variable is known to have a scope, . When you enter the function it comes alive and when you leave that function it is gone. But now we actually know what this physically means in hardware, . So, we would like to study the impact, or the effect of such an implementation on the security of this local variables. So, let us consider a simple example, . I have a function int fn that takes an int, and something, . And I have main where I am going to pass, I am going to call this function with the value 2, .

Now, let us assume that this function fn is doing some sort of an encryption, . You have a password that is being passed to it this integer n could be for example, some key that you are passing to it. Then you want to process this key and then figure out the password is correct or not send it to a database, get back the result see if it is correct or not. So, let us assume that this n, or the data that you are operating in this particular function, . So, let us assume that there are some local variables a b c and so on, . And, then I am doing some important operations here, and more than important they are highly critical or highly secure pieces of code , that need to be executed, .

So, let us look at by the way let us look at you know I have I am going to call this function fn of 2, and let us say there is an other function int fn underscore simple, which is not doing any critical operation, , but it does some other operation. So, I will call that simple, . So, what I am going to do now is to look at how the stack gets modified on calling this function in this particular sequence and see what happens to the local variables in fn in the process, . So, let me assume that the stack looks like this, my EBP is pointing arbitrarily here in main, and ESP is obviously moved up, because of the prologue in main.

So, now what do I want to do? I want to call this function of 2. So, what will I do, I will basically push 2 and call fn, . So, therefore, I will push the value two and my ESP will now move here and then I am going to call the function fn, and this is my return address by the way. After I finish fn I will return here address, right. So, the return address will get pushed onto stack here. And, then I am going to go into the function fn where of course, the prologue will get executed where I am going to move EBP of main, and then move my you know of course, in the process ESP would have got moved here and then I will move my EBP here, . So, EBP will get, EBP will come here and then my ESP will get, ok. This is my local variable space of fn, .

Now, the local variable a, b, c, d and all that stuff is simply going to exist here, . So, maybe I will just rub all of this out, I will just say this is the local variable a, b, c and so on, . So, I execute this function I operate on these variables a, b, c, d and all that and let us assume that somewhere on top you know on the topmost position. I have some variable z which is storing some important password, . Let us assume that this is some password value, ok. Let us assume it is an integer, for now.

So, now, I go ahead execute this function, I calculate this local variable z, I finish execution, which means that my epilogue will get executed, . What is the epilogue? It is going to add ESP with 0x maybe 40, bring ESP back here. Then it will pop EBP and cause my EBP to come back here, . Then I come to the return address, I come here and my ESP you know you do the stack cleanup and all that and then you come and call this function fn simple, .

So, now what is going to happen is this stack which was above main is now going to get over return. So, first of all before that note that after the function fn was executed and the stack pointer base pointer were all reset back to the values of main, this entire stack is really untouched, . The values are untouched, not touched, . Which implies that even after the function fn has executed and we have come back to main all those local variables simply hold their values as they are. It is just that because the EBP has been reset accessing EBP minus 4 or EBP minus 8, does not lead to accessing the local variables in fn, but it now accesses the local variables in main, . And this is the key point that I want to convey here when I talk about security of local variables.

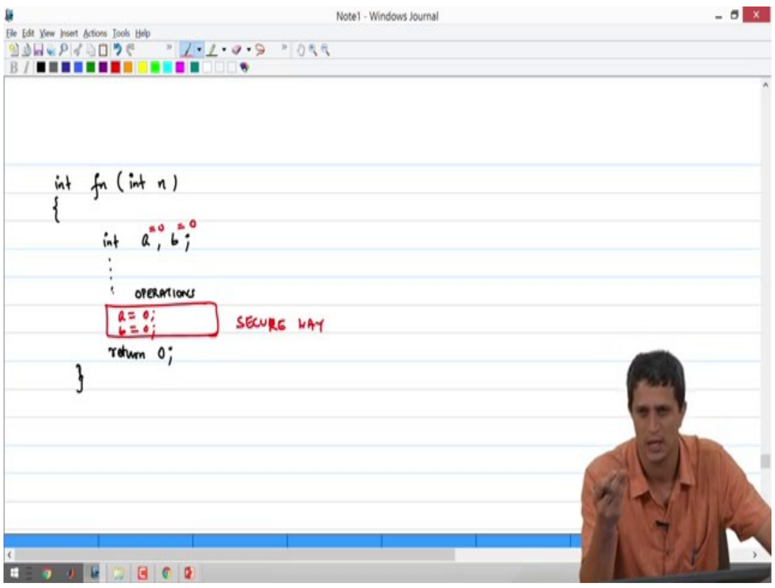
So, in spite of coming out of the function these local variable still have their values and these values if for example, happen to be a password then could be quite critical because they still exist in the stack, . So, if I do a dump of my program halfway through by just breaking into the code then I can literally see these values. So, if a hacker gets hold of your executable then by just breaking into the code I could see lot of values in this local variables and infer a lot of information, .

Now, for example, you may argue that, if I call fn simple, then the existing stack will get re return, which is true, right. When I call fn simple for example, then this return address will get modified to some other return address, . Of course now, I am not pushing anything on to stack therefore, instead of this I will do return address, . Then I will basically move my EBP of main into this particular location, EBP of main, right and then my local variable space of my function fn simple will happen here, , local variables of fn simple, .

So, the point I am trying to make is because you know this fn simple has a different number of parameters and so on, it is not necessary that the entire local variable space of fn will get over written in the process, only part of that may get overwritten and therefore, this local variable z may never get altered even though fn simple was called after that, . This may still, may still not get altered, . Therefore, you might find that certain important passwords or important encryption keys may be alive through the entire program in some local variable space on some stack, and by dumping all of this out it is possible to get hold of such important information.

Therefore, if you want to be absolutely sure that you have cleared out all the important information that was used in your program before you exited a particular function, then you have to ensure that before you leave that function use reset all these local variables to some other value, .

(Refer Slide Time: 11:33)



```
int fn (int n)
{
    int a, b;       (a =0, b =0)
    :
    operations
    a = 0;
    b = 0;          SECURE WAY
    return 0;
}
```

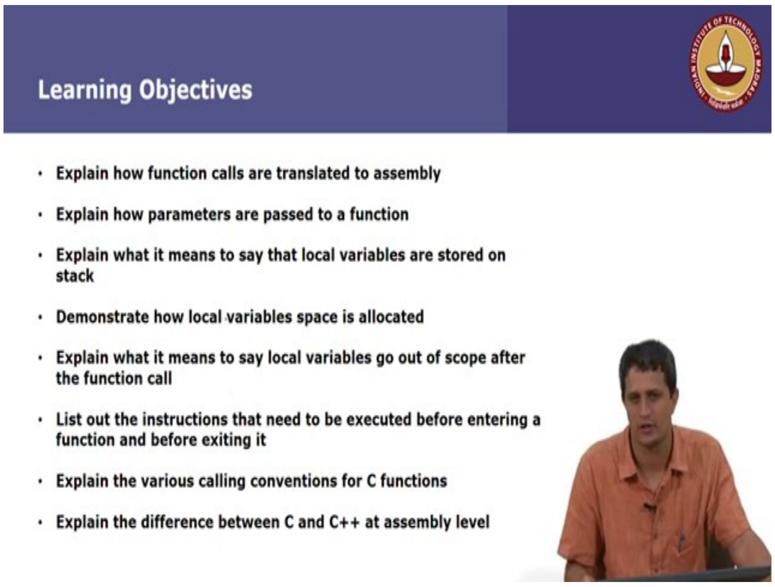
For example, if I have a function fn, of int n, int and I have local variables a b, and then you do some operations, . So, some operations, and then normally I would do a return 0, or some integer value. As we discussed now those local variables a and b will never get may never get erased through the entire life of the program, unless you call enough number of functions and so on, . It is there is a it is a matter of chance. Therefore, in order to improve the security of these local variables you could add some operations here, right for example, a equal to 0 and b equal to 0 before you exit this function.

So, now, what we are doing is before you come out of this function you are deliberately erasing the values that have been stored in a and b, and assuming these are critical pieces of information, you are resetting this local variables before you exit the function. And therefore, at any point if you now take a snapshot of your memory the local variable space will only be alive as long as you are in that particular function otherwise those local variables will always be 0. So, it is a very interesting result here.

We are used to initializing variables at entry point, for example, we are used to saying a equal to 0, and b equal to 0 when we start, but it is now interesting that even before you exit the function there are certain cases. Of course, there is a penalty of performance, I have to execute more number of instructions to complete the same function, but I can be safe and secure by ensuring that all important information in local variables have been erased before I exit that particular function, . So, this is a very secure way.

You could in fact, even modify the compiler, in order to generate this kind of a secure compilation, in order to generate such secure code you could alter the compiler itself to handle this automatically, you do not have to do it very on a case by case basis, . So, with that we have looked at you know a whole lot of things starting from module 1, the including the assembly language of 80 80, x 86 processors, then inline C, then you know how C program gets translated to assembly and certain the analysis of certain assembly translations, and what the impact of those translations are on performance, security and so on.

(Refer Slide Time: 14:53)



Learning Objectives

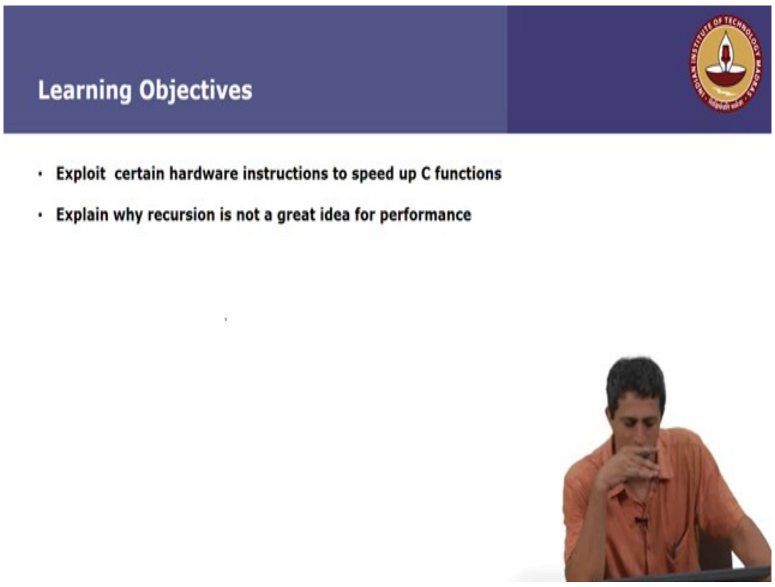
- Explain how function calls are translated to assembly
- Explain how parameters are passed to a function
- Explain what it means to say that local variables are stored on stack
- Demonstrate how local variables space is allocated
- Explain what it means to say local variables go out of scope after the function call
- List out the instructions that need to be executed before entering a function and before exiting it
- Explain the various calling conventions for C functions
- Explain the difference between C and C++ at assembly level

So, before we wind up I would like to just go through this learning objectives again and ensure that we have covered all of these topics in great detail. So, we started off with the learning objectives where we wanted to say that once you are done with the course as a student you should be able to explain how function calls are translated to assembly. We did a lot of this prologue, epilogue and so on. Explain how parameters are passed to a function? why you push parameters right to left, not left to right, you know they have to be pushed onto stack and so on.

What it means to say local variables are stored on stack, . Then demonstrate how local variable space is allocated, we saw which exact instruction allocates local variable space, . What it means to say local variables goes out of scope after the function call, . We in fact, just now in this lecture saw that going out of scope does not mean the value is gone, the value still remains and can be potentially a security threat and to overcome that you have to reset those variables before you exit the function, .

Then listing out instructions that need to be executed before entering and before exiting the function the prologue and epilogue, is something we looked at in great detail. Various calling conventions, how stack cleanup happened whether it inside the function outside the function and so on. Then the difference between C and C plus plus, right; in C plus plus you have to pass this pointer through the ECX register, . So, this is an other important difference between C and C plus plus at assembly level, . At a programming and syntactical level there are lots of differences, but assembly level this is only difference, .

(Refer Slide Time: 16:37)



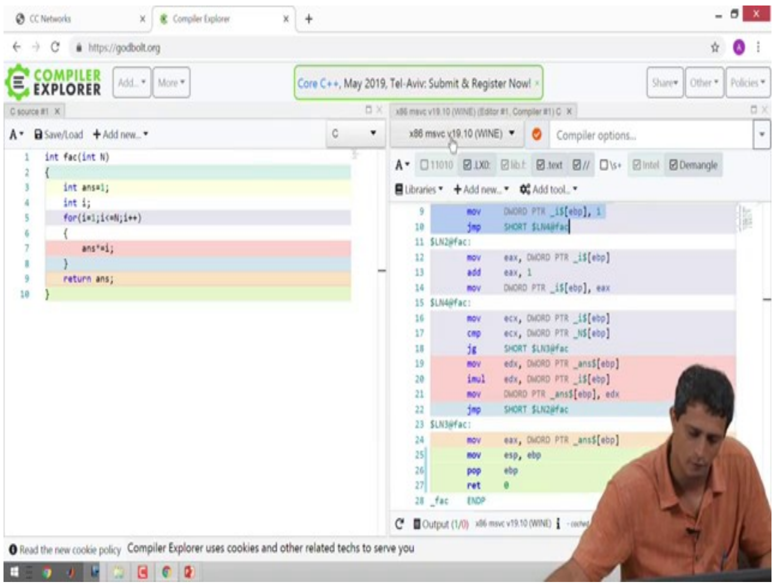
Learning Objectives

- Exploit certain hardware instructions to speed up C functions
- Explain why recursion is not a great idea for performance

And then we wanted to look at exploiting certain hardware instructions to speed up C functions like memcpy, string length and so on, . And then we wanted to look at why recursion is not a great idea for performance. So, even there we saw that the prologue and epilogue and function parameter passing is a significant overhead to the instruction that you eventually want to implement in the function and therefore, it is not a great idea to do recursion when performances important, .

So, with that we come to the end of this course. And I have taken you through a very physical implementation, . I would like to stress here that what I discussed in this course is not the only way that C programs can be translated to assembly language, .

(Refer Slide Time: 17:35)



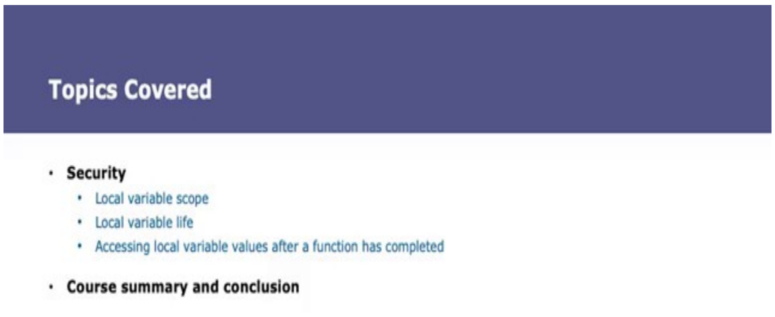
```c
int fac(int N)
{
    int ans=1;
    int i;
    for(i=1;i<=N;i++)
    {
        ans*=i;
    }
    return ans;
}
```

```asm
        mov     DWORD PTR _i$[ebp], 1
        jmp     SHORT $LN4@fac
$LN2@fac:
        mov     eax, DWORD PTR _i$[ebp]
        add     eax, 1
        mov     DWORD PTR _i$[ebp], eax
$LN4@fac:
        mov     ecx, DWORD PTR _i$[ebp]
        cmp     ecx, DWORD PTR _N$[ebp]
        jg      SHORT $LN3@fac
        mov     edx, DWORD PTR _ans$[ebp]
        imul    edx, DWORD PTR _i$[ebp]
        mov     DWORD PTR _ans$[ebp], edx
        jmp     SHORT $LN2@fac
$LN3@fac:
        mov     eax, DWORD PTR _ans$[ebp]
        mov     esp, ebp
        pop     ebp
        ret     0
_fac    ENDP
```

So, if you go look at the this compiler explorer here, you have a whole lot of you know you know arm 64 you know let us look at you know something simpler with x, . There some problem with this tool, right now, . But, you see here that there are a whole lot of assembler and compiler options. And if you look at the compiler output from each of these compilers it might be a little different, right, but the heart of what I have thought here is still applicable even to those compilers and that compiled output therefore, it should be possible, if you understand the instruction set of that microprocessor it should be very easy for you to understand what is going on, .

The idea of allocating local variable space, pushing parameters on to stack, storing EBP on stack, all of this will be common will be done in some way or the other. It need not be done exactly in the order that was mentioned here in msvc, . ah So, with that I hope as a student you can now go ahead and use this compiler explorer tool and also the material that I have taught, in this course in order to explore compilers more easily and I hope the compiler output does not intimidate students anymore.

(Refer Slide Time: 19:13)



Topics Covered

- **Security**
  - Local variable scope
  - Local variable life
  - Accessing local variable values after a function has completed
- **Course summary and conclusion**

Thank you.