ENEE 457: Computer Systems Security

Lecture 16
Buffer Overflow Attacks

Charalampos (Babis) Papamanthou

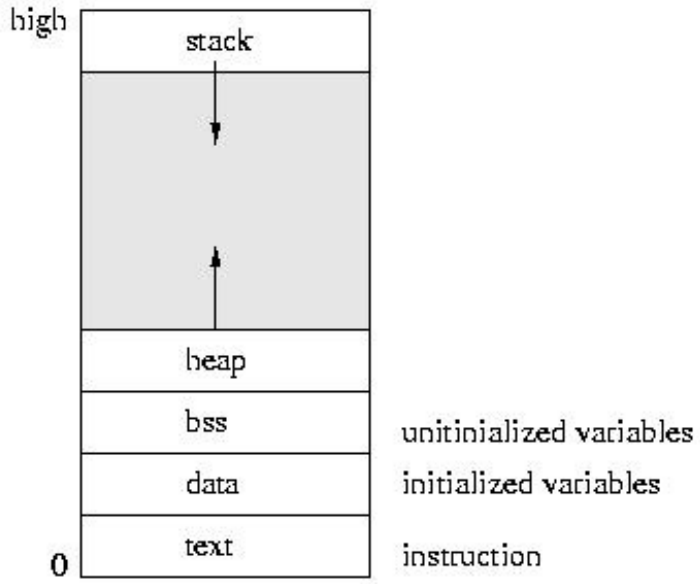
Department of Electrical and Computer Engineering
University of Maryland, College Park

Buffer overflow attacks

- We have already seen some kind of a buffer overflow…
- Heartbleed
- Buffer overflow attacks lead to undesired behavior (e.g., in Heartbleed they led to the secret key being output to the attacker) when the attacker causes the system to read data outside an “eligible” memory area

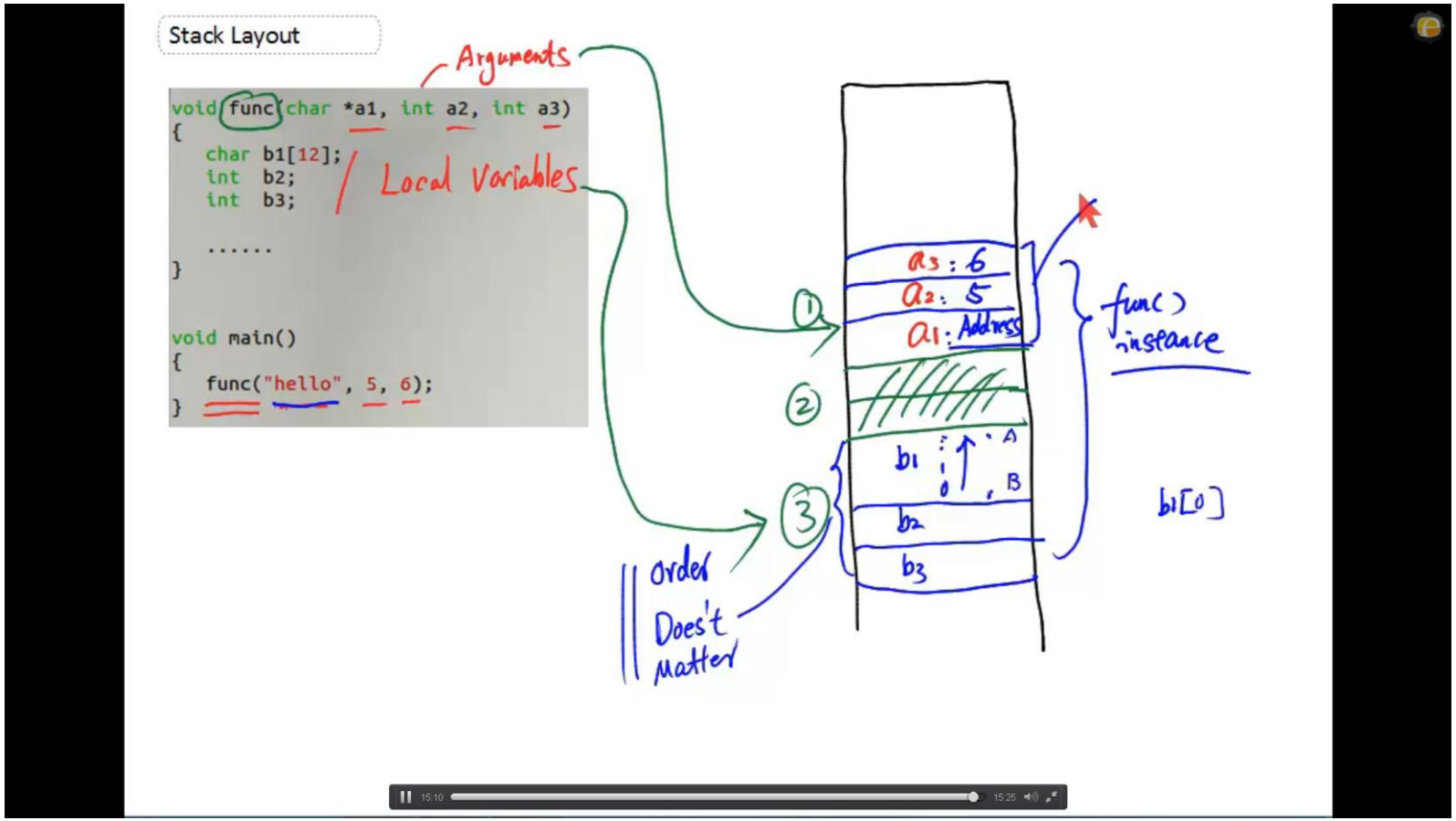
Memory Layout

- Stack and heap and the dynamically allocated portions of memory whose size changes while program runs
- Stack is used for keeping track of functions calls, heap is used for allocating dynamic memory (note that these cannot be predermined at compilation time)



Stack Layout

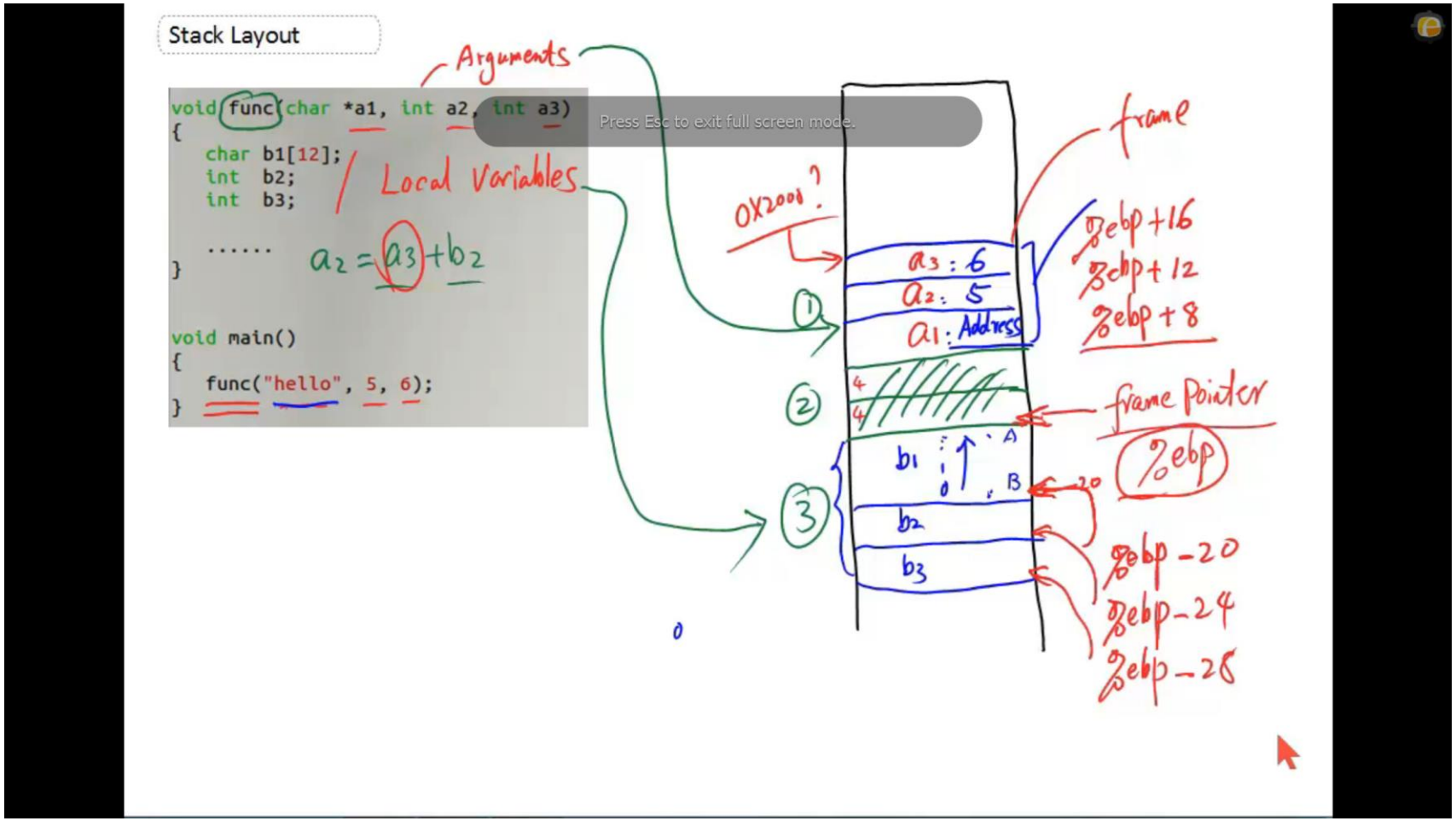
- We store arguments at the top of the stack frame and other variables at the bottom of the stack frame



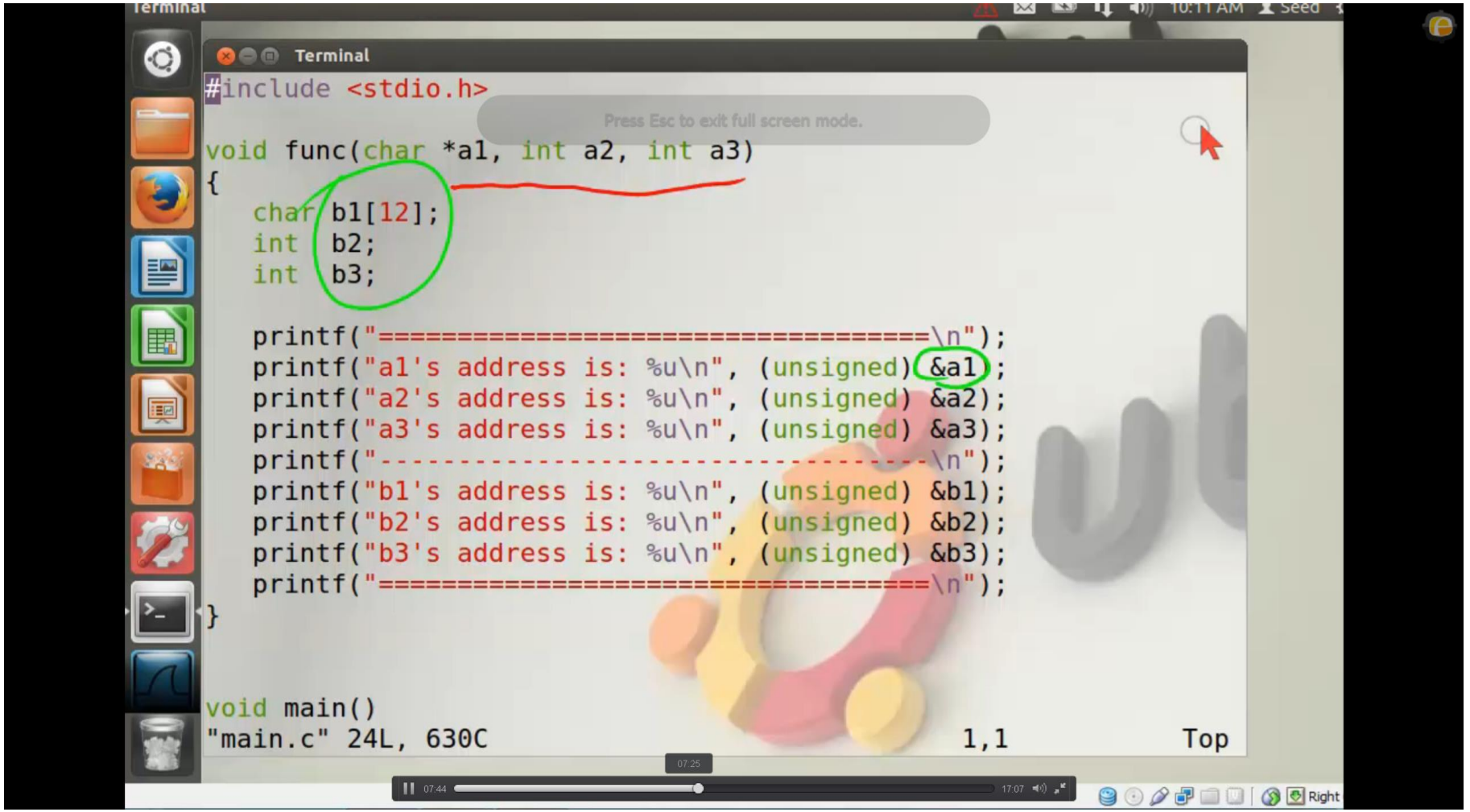
Addresses of variables during runtime

- The CPU needs to access addresses of local variables in the function at runtime
- Unfortunately, these addresses are not known at compilation time (at compilation time, I only know the addresses relative to the top of the stack frame)
- To deal with that, there is a system register ebp which stores the address of the current stack frame (basically the top of the stack)
- So whenever CPU needs to access a specific local variable within a stack frame, it just uses ebp and the relative offset

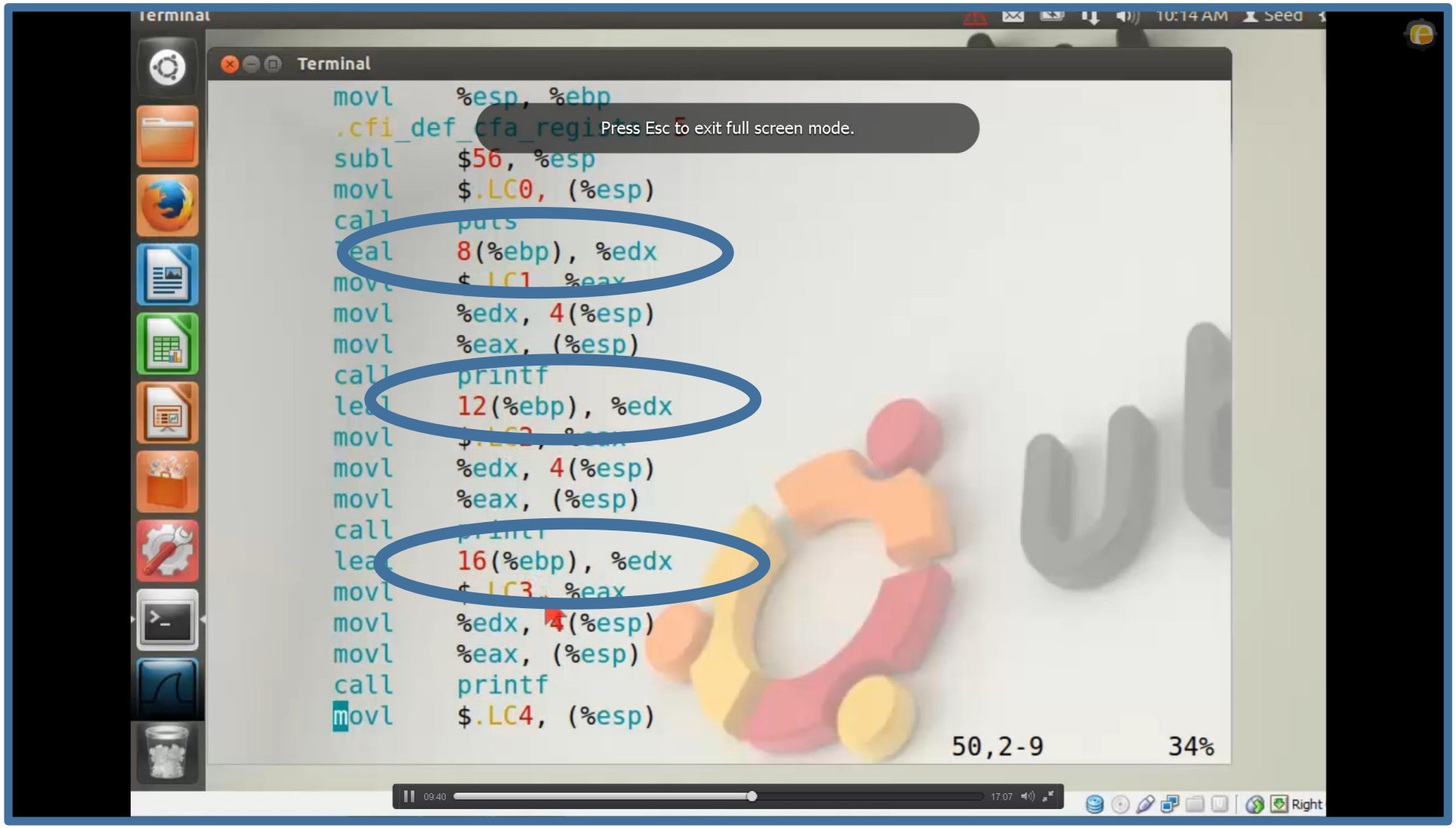
Example


Stack Layout
Arguments
Local Variables
Press Esc to exit full screen mode.
void func(char *a1, int a2, int a3)
{
char b1[12];
int b2;
int b3;
.....
}
a2 = a3 + b2
void main()
{
func("hello", 5, 6);
}
0X2001?
frame
a3 : 6
a2 : 5
a1 : Address
%ebp+16
%ebp+12
%ebp+8
frame Pointer
%ebp
b1
b2
b3
%ebp-20
%ebp-24
%ebp-28
1
2
3

Let’s see the assembly code of main.c



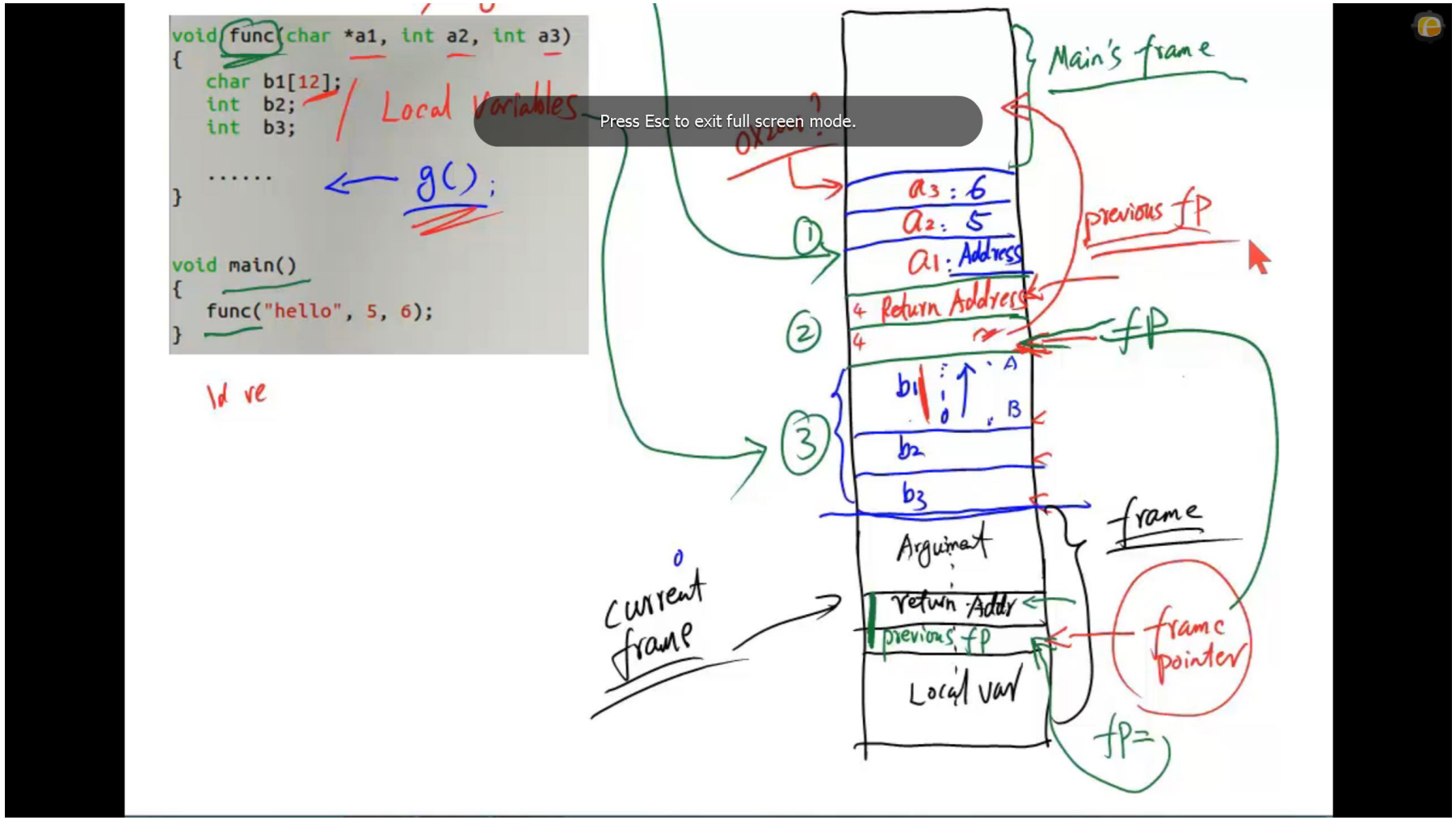
Assembly


Terminal
Terminal
Press Esc to exit full screen mode.
movl %esp, %ebp
subl $56, %esp
movl $.LC0, (%esp)
call puts
leal 8(%ebp), %edx
movl %edx, 4(%esp)
movl %eax, (%esp)
call printf
leal 12(%ebp), %edx
movl %edx, 4(%esp)
movl %eax, (%esp)
leal 16(%ebp), %edx
movl %edx, 4(%esp)
movl %eax, (%esp)
call printf
movl $.LC4, (%esp)
50,2-9
34%
Right

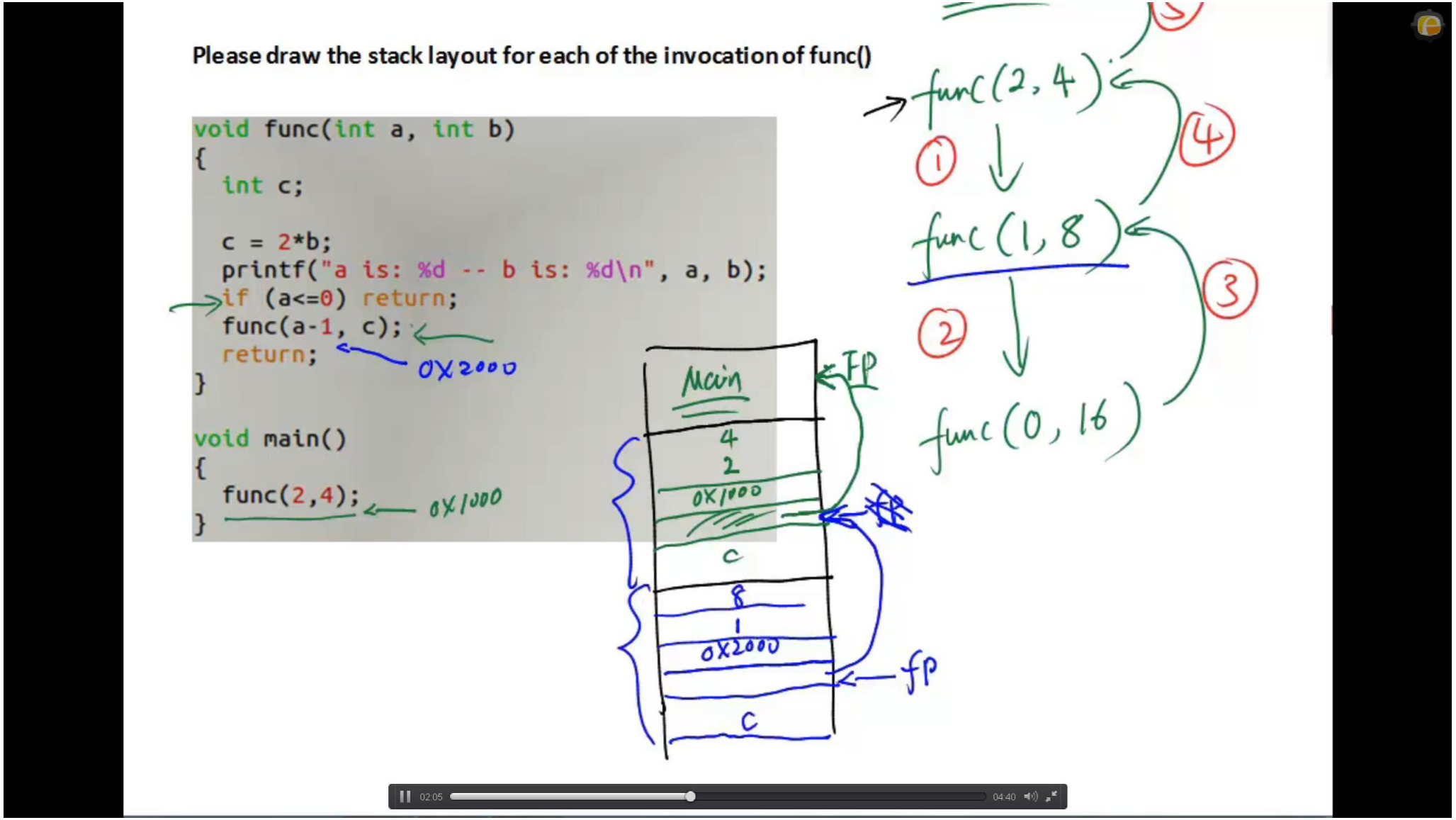
Return address and previous frame pointer

- Between the arguments and the local variables, the stack frame will store the return address, namely, where the execution needs to return after the function call is done and releases the memory
- Also, it will store the previous frame pointer, so that the ebp can be updated whenever the execution is done

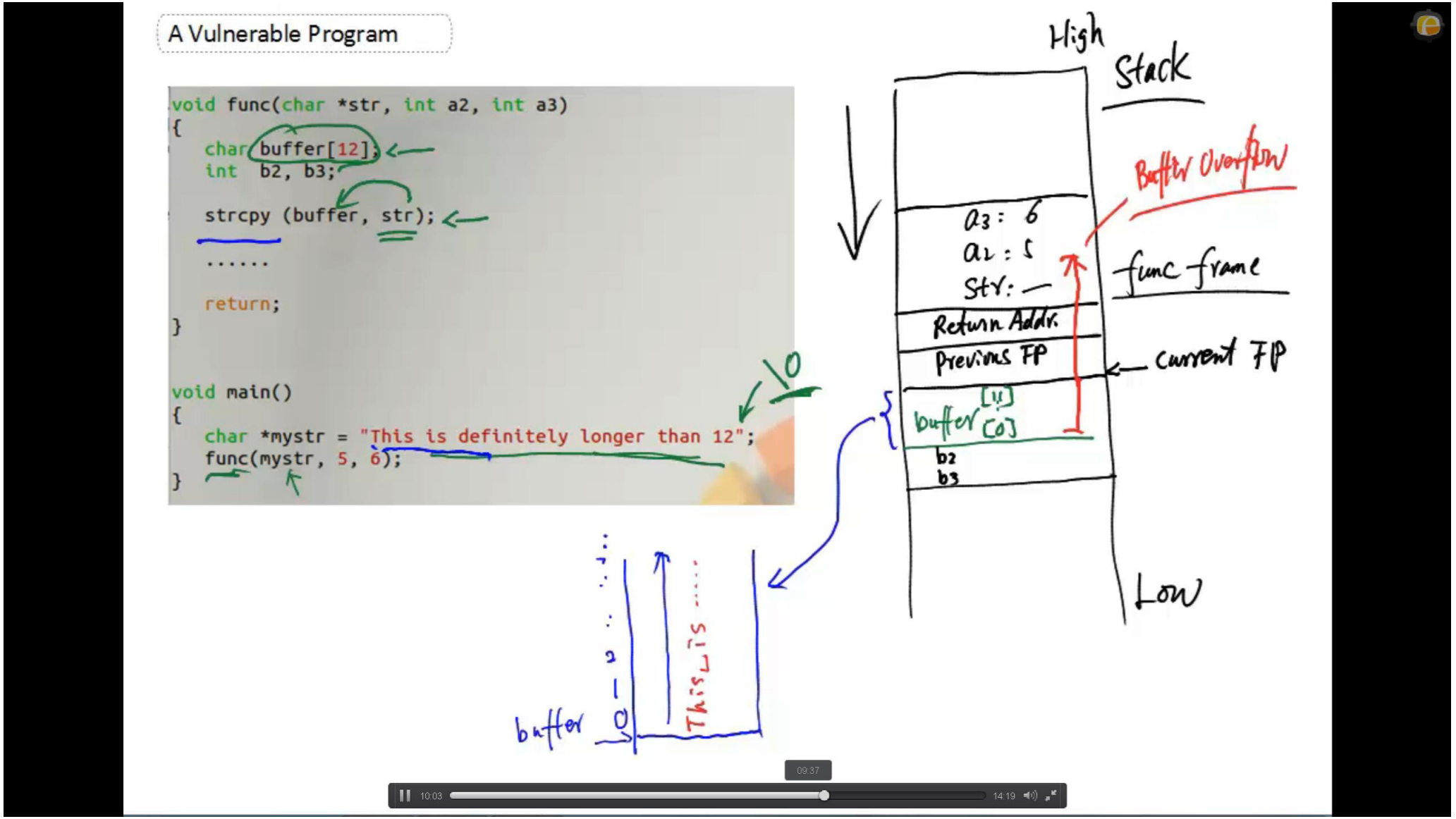
Example


void func(char *a1, int a2, int a3)
{
char b1[12];
int b2;
int b3;
.....
}
void main()
{
func("hello", 5, 6);
}
Local variables
g();
Press Esc to exit full screen mode.
Main's frame
previous fp
fp
frame
frame pointer
current frame
Argument
Return Addr
previous fp
Local var
fp=

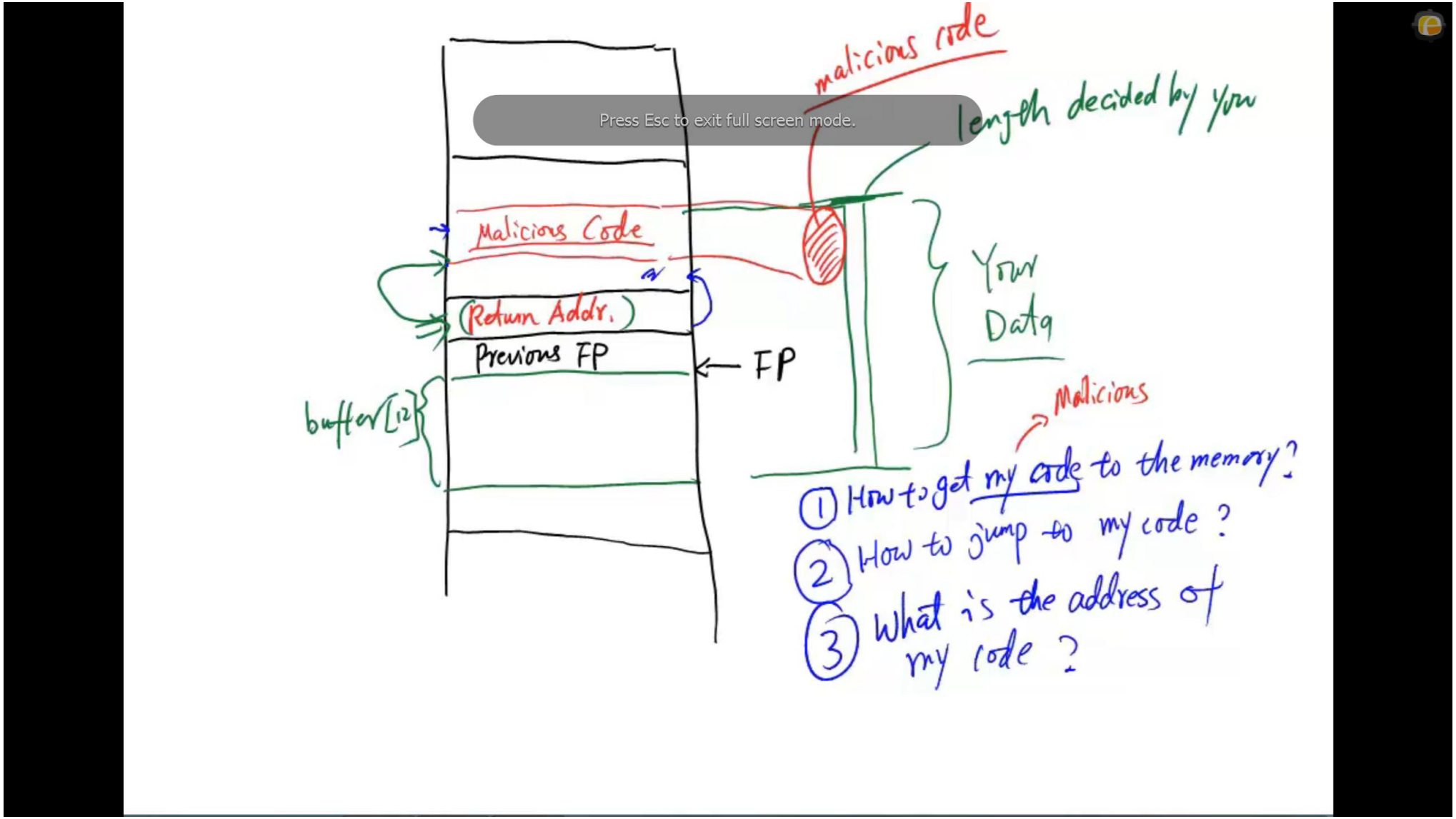
Drawing the stack layout for a recursive program



Basics of Buffer Overflow attack: Example



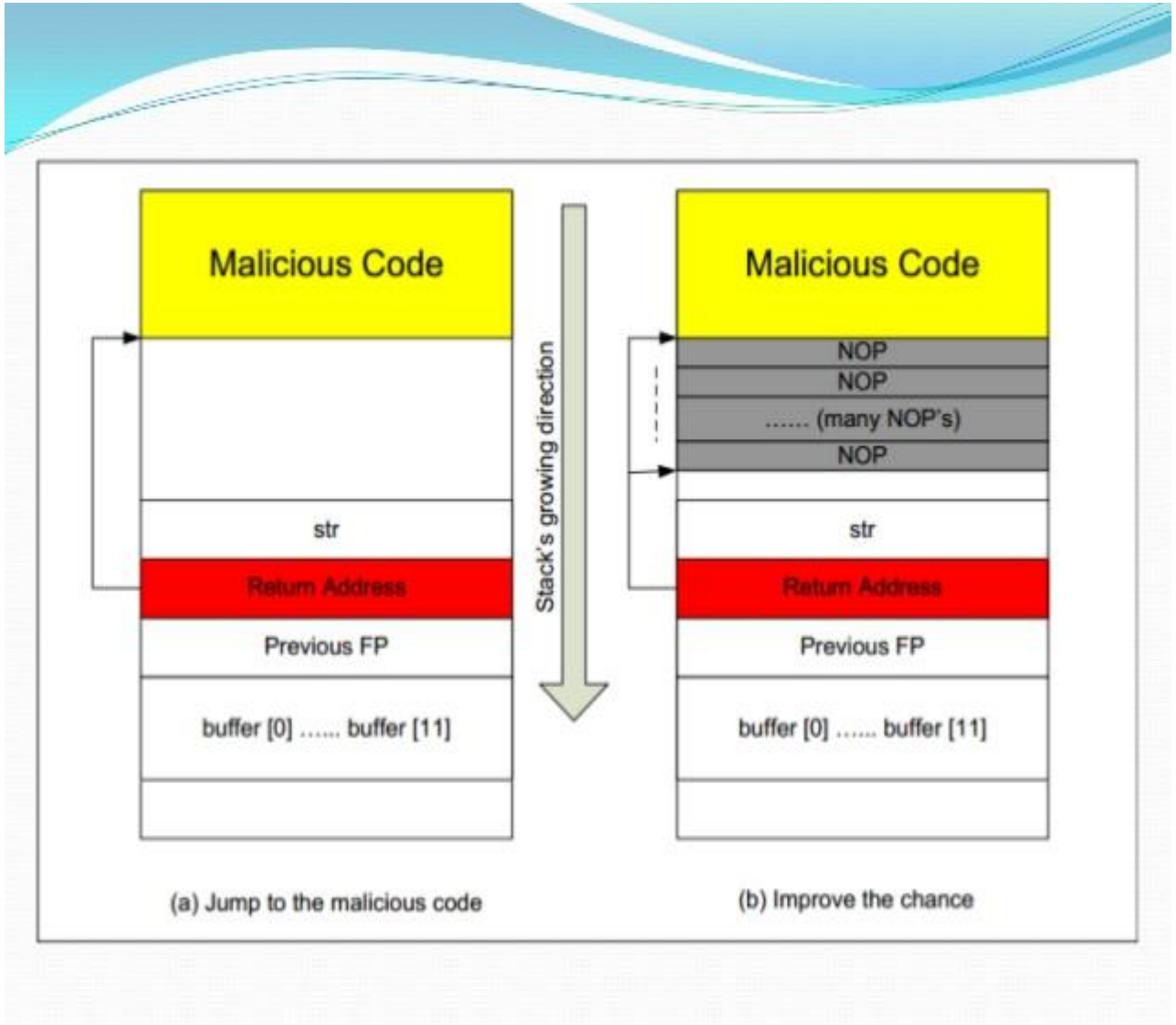
Challenges of the attack



How to find the return address?

- The attacker can use FP to figure out what address he should be using to overwrite the return address and place the malicious code
- But FP is not printed by the program. This is not likely
- Possible solutions:
  - Try many choices (low probability of success)
  - Try to run a similar/same program to see how the addresses are distributed. In particular if you do not randomize the addresses of the variables on the stack, the addresses you will be getting on your own execution will be similar to the addresses of the program you are trying to attack, since the stack is always starting from a fixed location
- Note also that stack does not grow that much (unless you are doing a lot of recursion) so you can hit the right address with very high probability

Find the return address: Using NOOP



What program to run?

```
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x8
8\x46\x07\x89\x76\x08\x
89\x46"
"\x0c\xb0\x0b\x8d\x1e\x8
d\x4e\x08\x8d\x56\x0c\x
cd\x80"
"\xe8\xe3\xff\xff\xff\x2f\x6
2\x69\x6e\x2f\x73\x68";
```

Challenge #1

- How to estimate the distance from the beginning of the buffer to the address where the return address is stored?
  - Just debug and print the address x of the beginning of the buffer and the address of %ebp
  - Remember %ebp will point to one position behind the return address
  - So the distance is %ebp – x+4

Challenge #2

- What address to overwrite the return address with
  - Any value greater than %ebp + 8 will do
  - This is because you can place the malicious code at any point in memory after the return address and the return address is finishing 8 bytes after %ebp