

IT3789 Cyber Security Attack & Defence



L09 - Buffer Overflow Exploitation

**WITH KNOWLEDGE
COMES RESPONSIBILITY**

Buffer Overflow Exploitation

**Win32
Assembly**

Fuzzing

**Buffer
Overflows**

**Stack
Exploitation**

**Heap
Exploitation**

**Exploit
Protection
Mechanism**

Application Memory Layout

- Every application is assigned 4GB (2^{32} bytes) of virtual memory space.
 - Address range: 0x00000000 to 0xFFFFFFFF
 - User mode: Lower 2GB of address space
 - Memory area where application is loaded and executed.
 - No direct access to hardware and only restricted access to memory.
 - Kernel mode: Upper 2GB of address space
 - Memory area where kernel mode components (e.g. kernel32.dll, ntdll.dll) are loaded and executed.
 - Direct access to hardware and memory. (Privilege mode)

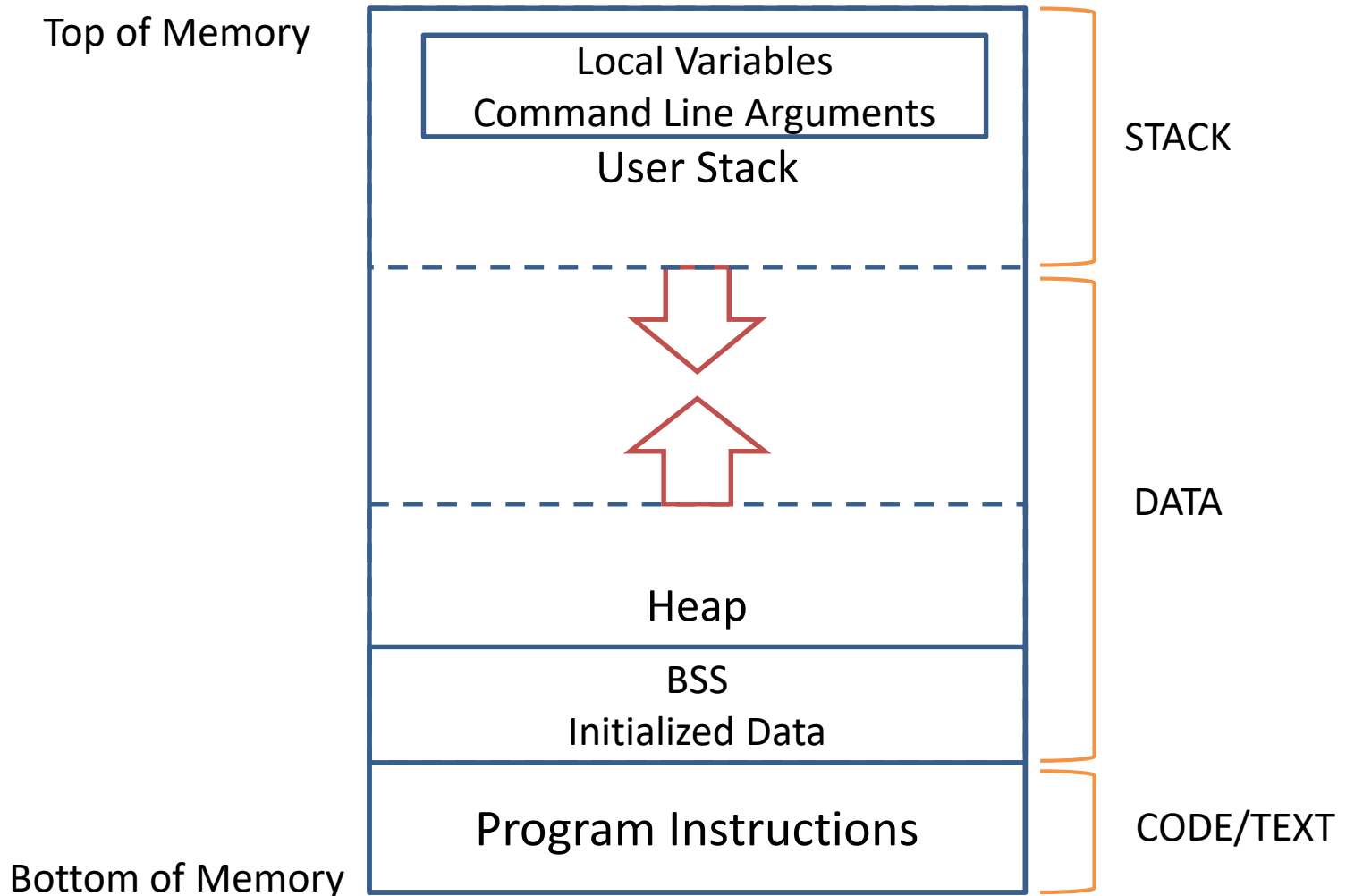
Application Memory Layout

- The virtual address is mapped to physical addresses where the data exists when the application executes.
 - The physical address space is divided into 4KB pages in the x86 processor.
- An application has no direct access to kernel mode memory.
 - Prevents applications from damaging the system or its features.
 - Any attempts to do so will cause an access violation.
 - Switching from user to kernel mode allows application to have proper access to the kernel.
 - Via system calls or interrupts for certain events such as timers, keyboard and hard disk I/O.
 - Switches back to user mode after processing is done.

Process Memory Organization

- Application processes are loaded into 3 major memory areas.
 - Stack Segment
 - Data Segment
 - Code/Text Segment
- Data and stack segment are private to each application.
- Code/Text segment is read-only and can be accessed by other processes.

Process Memory Organization



Process Memory Organization

| Memory Area | Stores |
|-------------------|------------------------------------------------------------------------------------------------|
| Stack Segment | <ul style="list-style-type: none">• Local variables• Procedure calls |
| Data Segment | <ul style="list-style-type: none">• Static variables• Dynamic variables |
| Code/Text Segment | <ul style="list-style-type: none">• Program instructions |

Buffer Overflow Exploitation

**Win32
Assembly**

Fuzzing

**Buffer
Overflows**

**Stack
Exploitation**

**Heap
Exploitation**

**Exploit
Protection
Mechanism**

Fuzzing

- Method of finding flaws in a software by providing unexpected inputs and monitoring for exceptions.
- Typically an automated or semi-automated process.
 - Manipulating and supplying data to target software for processing.

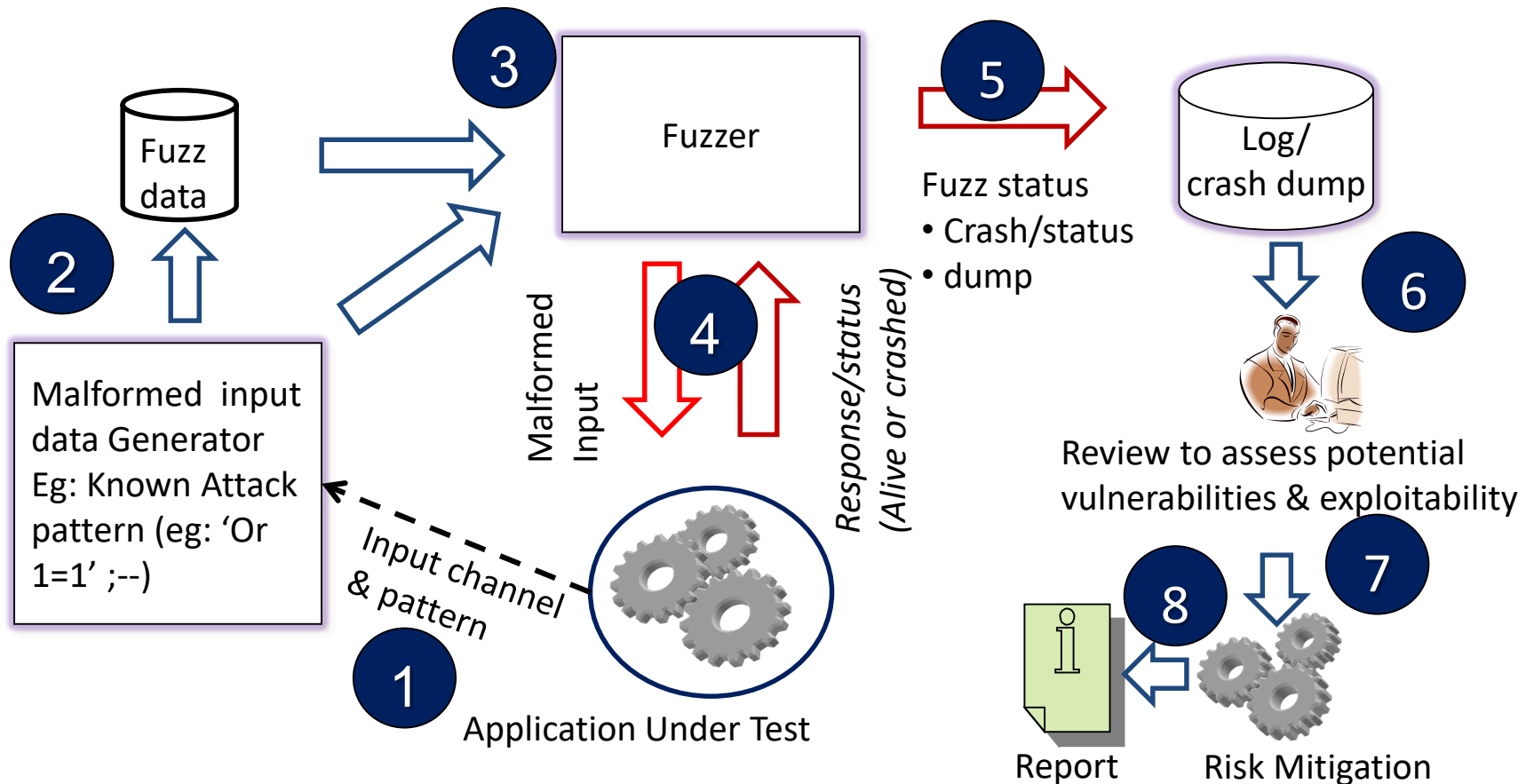


Fuzzing

- Fuzzers typically target inputs that cross a trust boundary as these are more easily accessible.
- The key is to find input combination not expected by the program.

General Fuzzing Process

Fuzzing is a black box security/reliability testing technique.



Fuzz Tools

- Microsoft Security Risk Detection (binary fuzzing in the cloud) - <https://www.microsoft.com/en-us/security-risk-detection/>
- SPIKE – protocol fuzzer (in Kali or <https://github.com/guilhermeferreira/spikepp>)
- Peach (<http://peachfuzzer.com>)
- Fuzzers from OWASP (WebScarab, for HTTP and HTTPS protocols; WSFuzzer, for SOAP protocol)

Buffer Overflow Exploitation

**Win32
Assembly**

Fuzzing

**Buffer
Overflows**

**Stack
Exploitation**

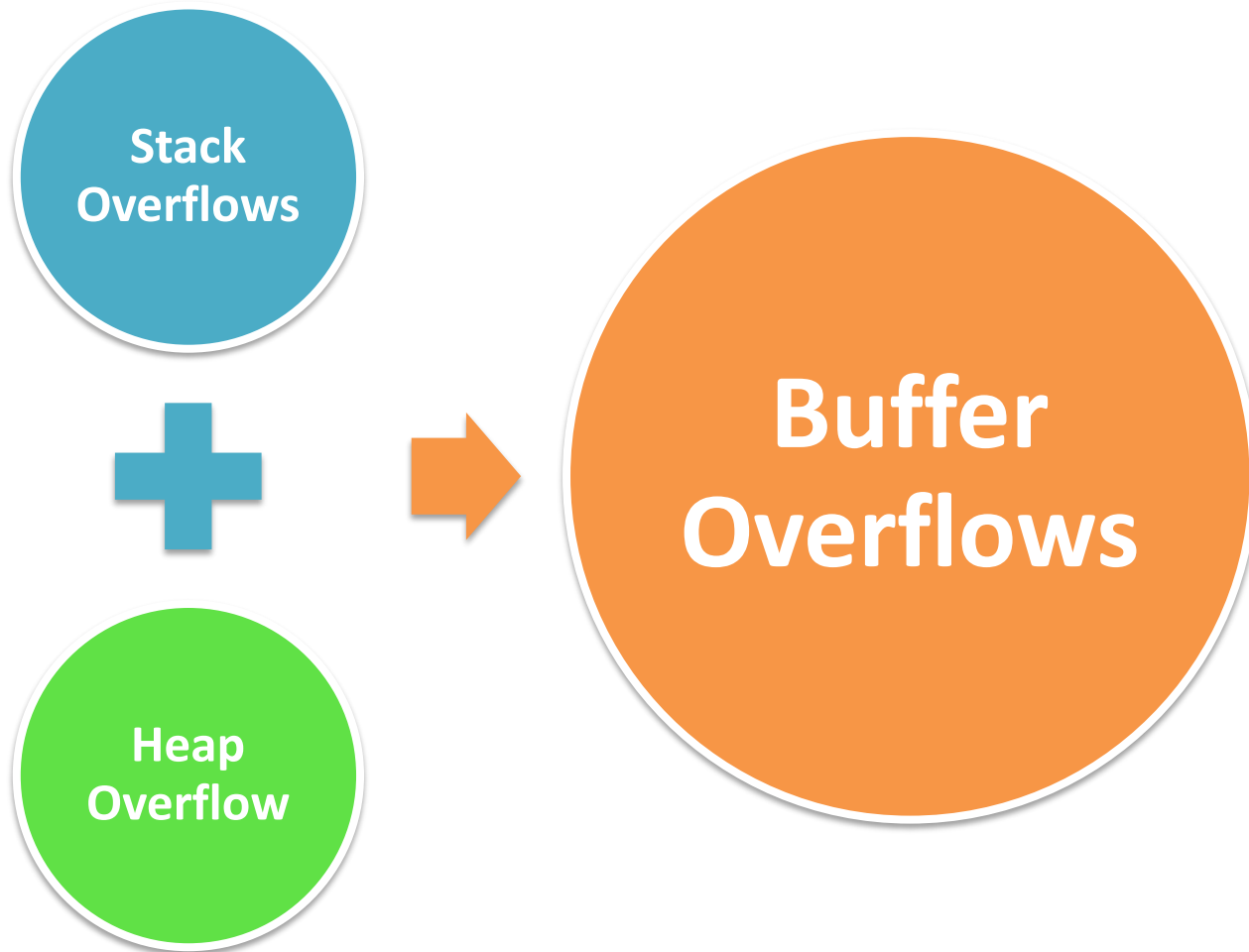
**Heap
Exploitation**

**Exploit
Protection
Mechanism**

Buffer Overflows

- When buffer overflows happen?
 - When input assigned to a variable is larger than the memory allocated for that variable.
 - Main cause: Programmers overlooked what happens after data overflows the memory that has been allocated.
- Hackers can use buffer overflow techniques to exploit these overflows.

Types of Buffer Overflows



Buffer Overflow Exploitation

**Win32
Assembly**

Fuzzing

**Buffer
Overflows**

**Stack
Exploitation**

**Heap
Exploitation**

**Exploit
Protection
Mechanism**

The Stack Memory

- LIFO (Last in first out) structure.
 - Mainly for storage of local variables and data related to function calls.
- Important registers for stack operation.
 - ESP: Extended Stack Pointer
 - Points to the top of the stack.
 - EBP: Extended Base Pointer (Frame Pointer)
 - Value in this register usually stays the same throughout a function execution.
 - Use to reference stack-based information such as arguments and local variables in a function using offset.
 - EIP: Extended Instruction Pointer
 - Determine the location of the next instruction to be executed.
 - Saved on the stack when a function is called.

Stack Frame

- Stack frame is the entire stack section that is used for a function. (Define by ESP and EBP)
- CALL and RET instructions combination allows code to be jumped to and returned.
 - CALL instruction
 1. Push address of the next instruction in EIP to the stack.
 2. Jump to the address specified by the call.
 3. Execute function prologue.
 - RET instruction
 1. Execute function epilogue.
 2. Pop the stored address off the stack to EIP.
 3. Jump to the address in EIP.

Function Prologue

- A function prologue is called at the beginning of a function.
 - Prepare the stack and registers for use within the function.
- It typically does the following actions.
 1. Pushes the current base pointer (EBP) onto the stack.
 2. Replaces the old base pointer (EBP) with the current stack pointer (ESP).
 3. Moves the stack pointer make room for the function's local variables. (if any)
- Instructions

```
push ebp      ; preserve the caller stack frame
mov  ebp, esp ; prepare new stack frame for callee
sub  esp, n    ; n is the size of the local variables (in bytes),
               reserve memory space for local variables
```

Function Epilogue

- A function epilogue returns control to the calling function.
 - Called at the end of the function.
 - Reverse of function prologue.
 - Restores the stack and registers to the state before the function was called.
 1. Replaces the stack pointer (ESP) with the current base pointer (EBP).
 2. Pops the base pointer off the stack to EBP.
 3. Returns to the calling function.
- Instructions

```
mov esp, ebp    ; restore caller stack frame
pop ebp         ; restore caller stack frame
ret n           ; n is the size of the local variables (in bytes),
                ; clean up memory space for local variables
```

Example Function

```
void called(int a, int b, int c) {  
    char buffer1[8];  
    char buffer2[12];  
    ...  
}
```

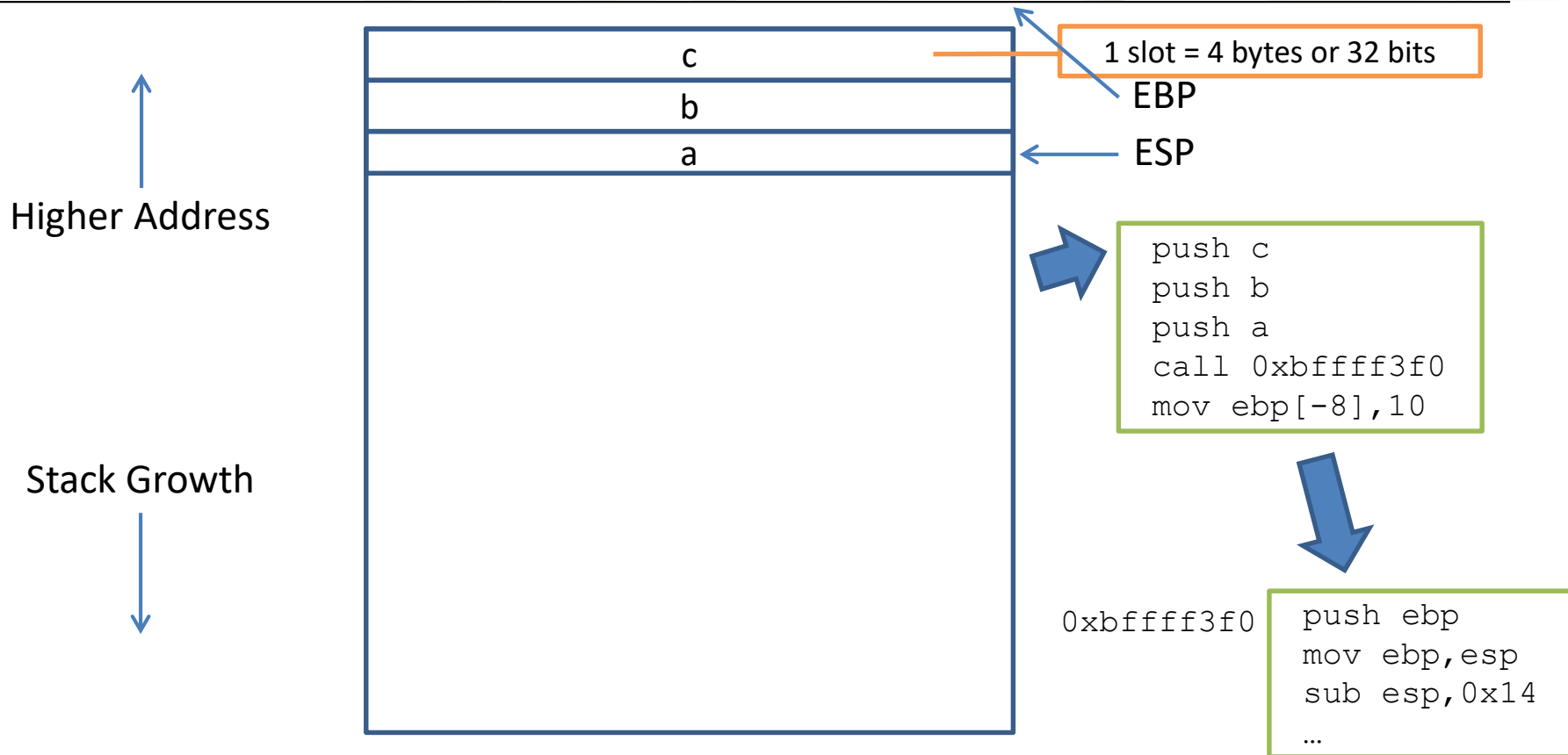
```
void caller() {  
    called(1,2,3);  
    int x = 10;  
}
```



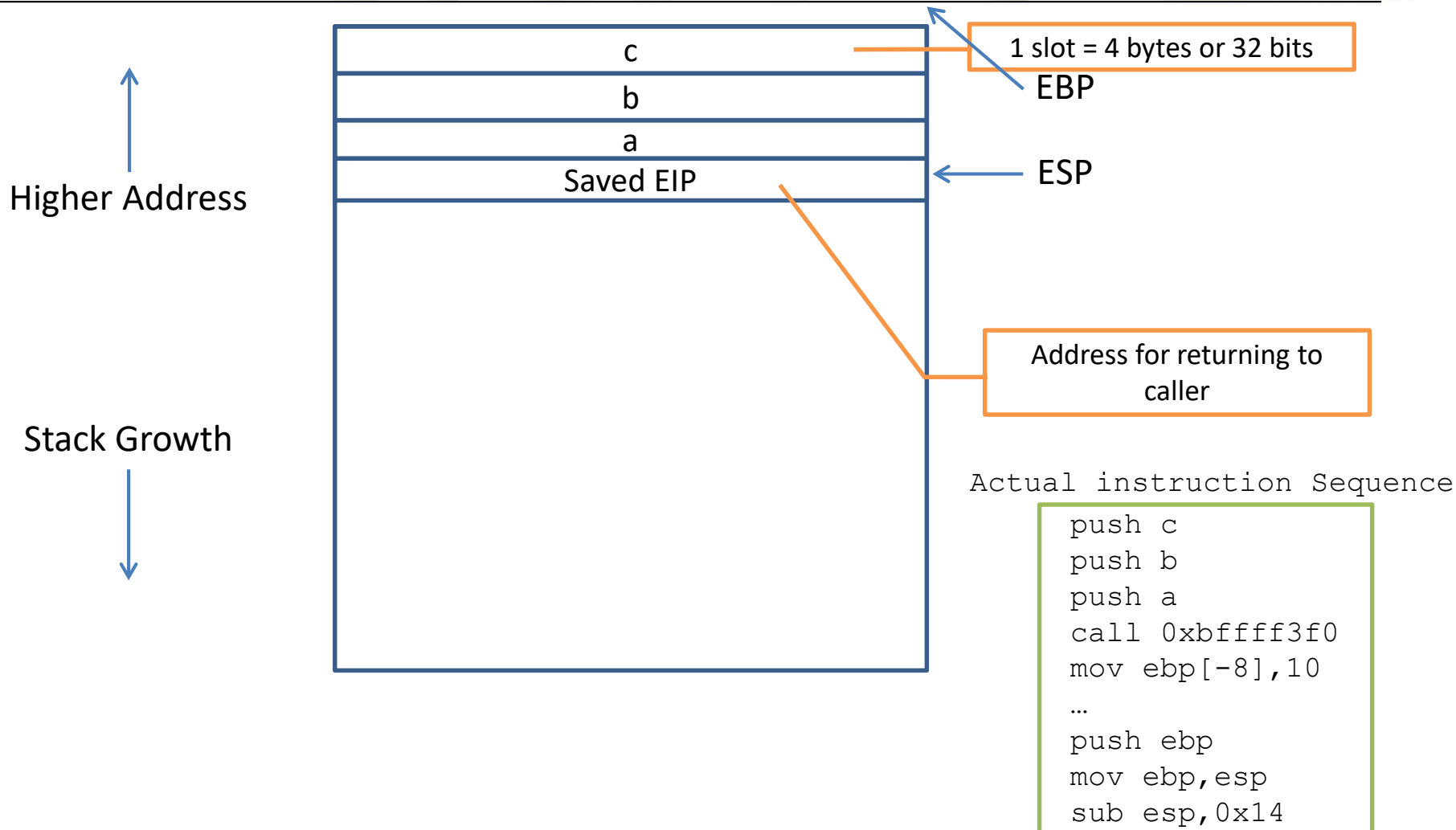
```
push c  
push b  
push a  
call 0xbffff3f0  
mov ebp[-8],10
```

Memory address to the called()

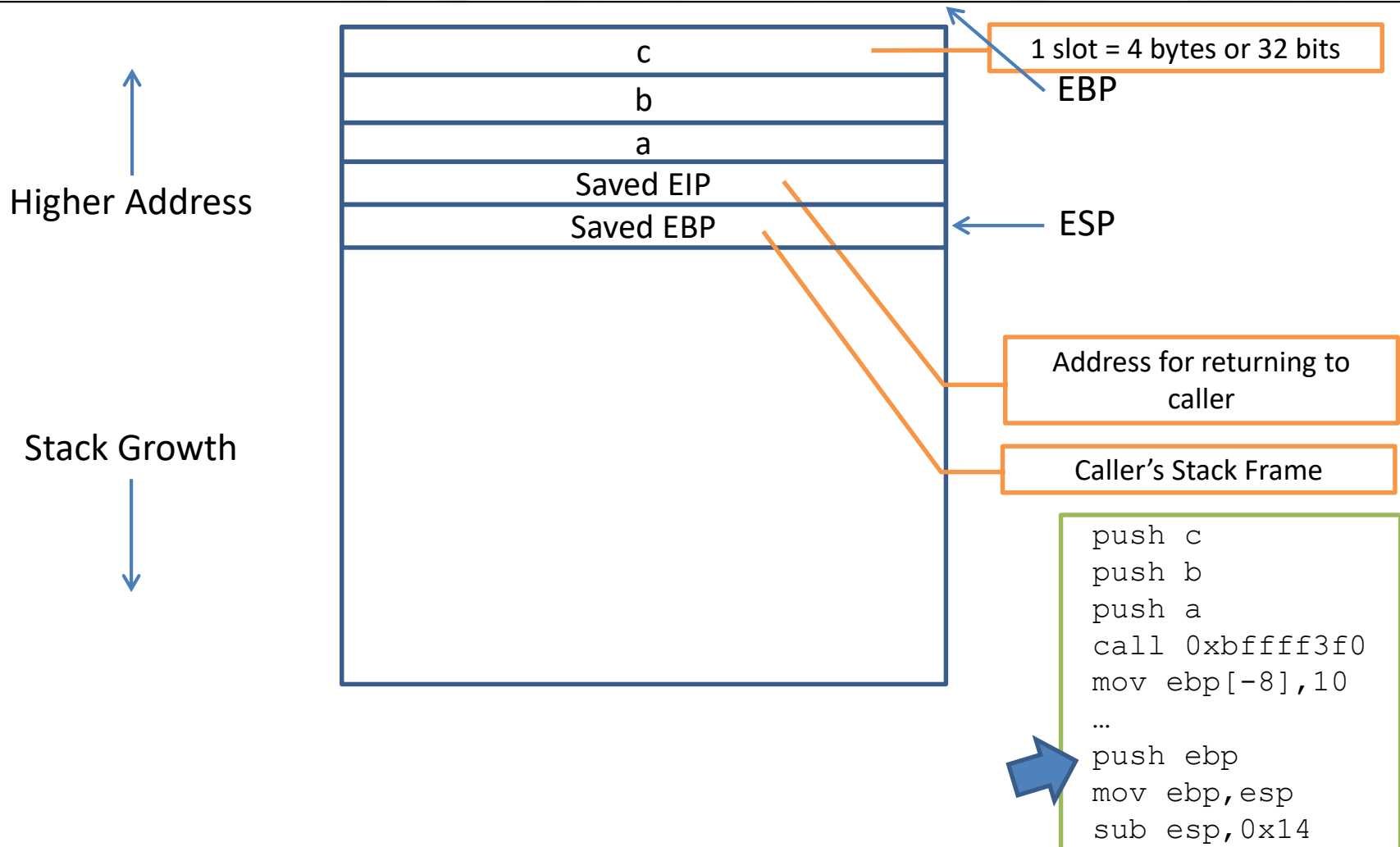
Function Call



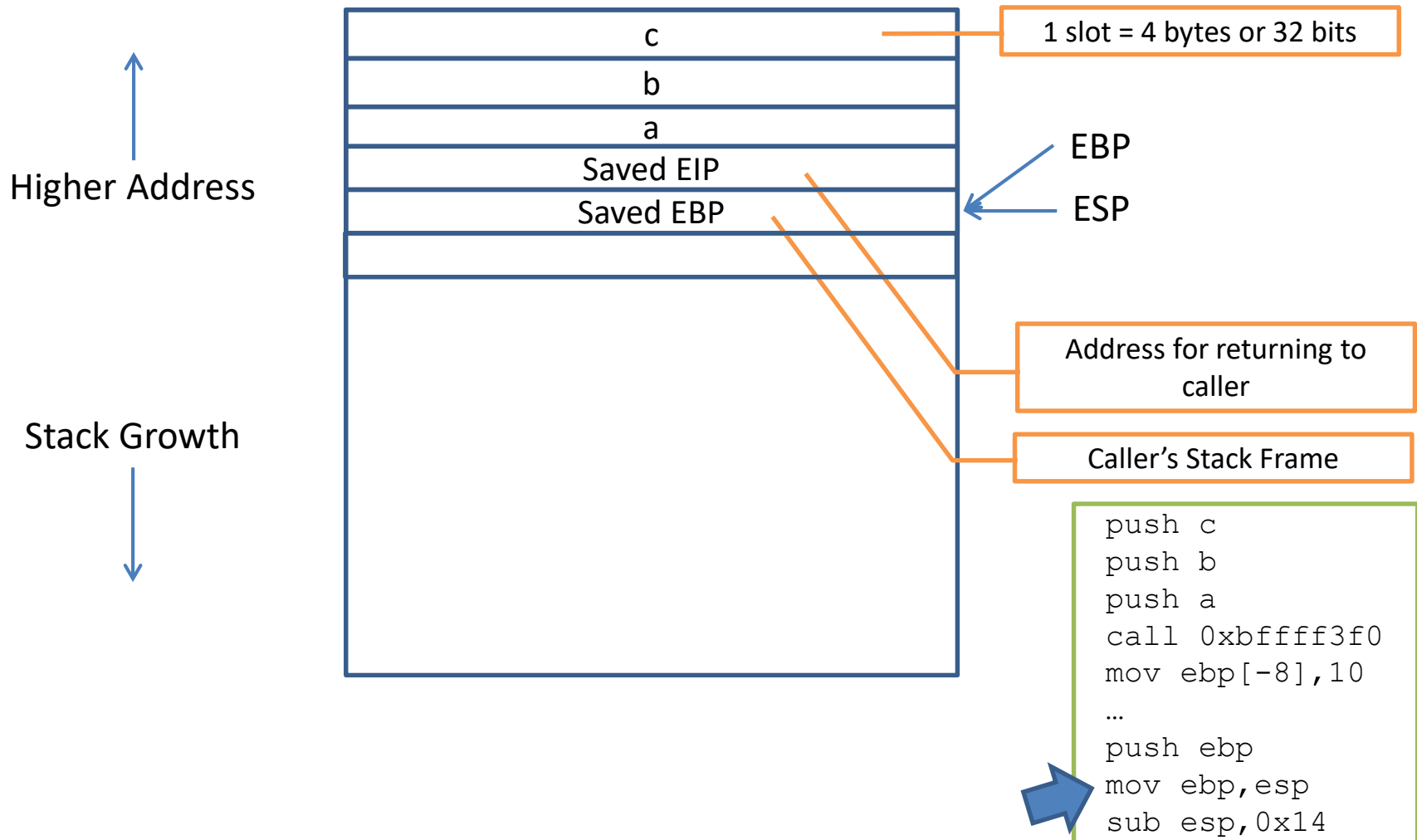
Function Call



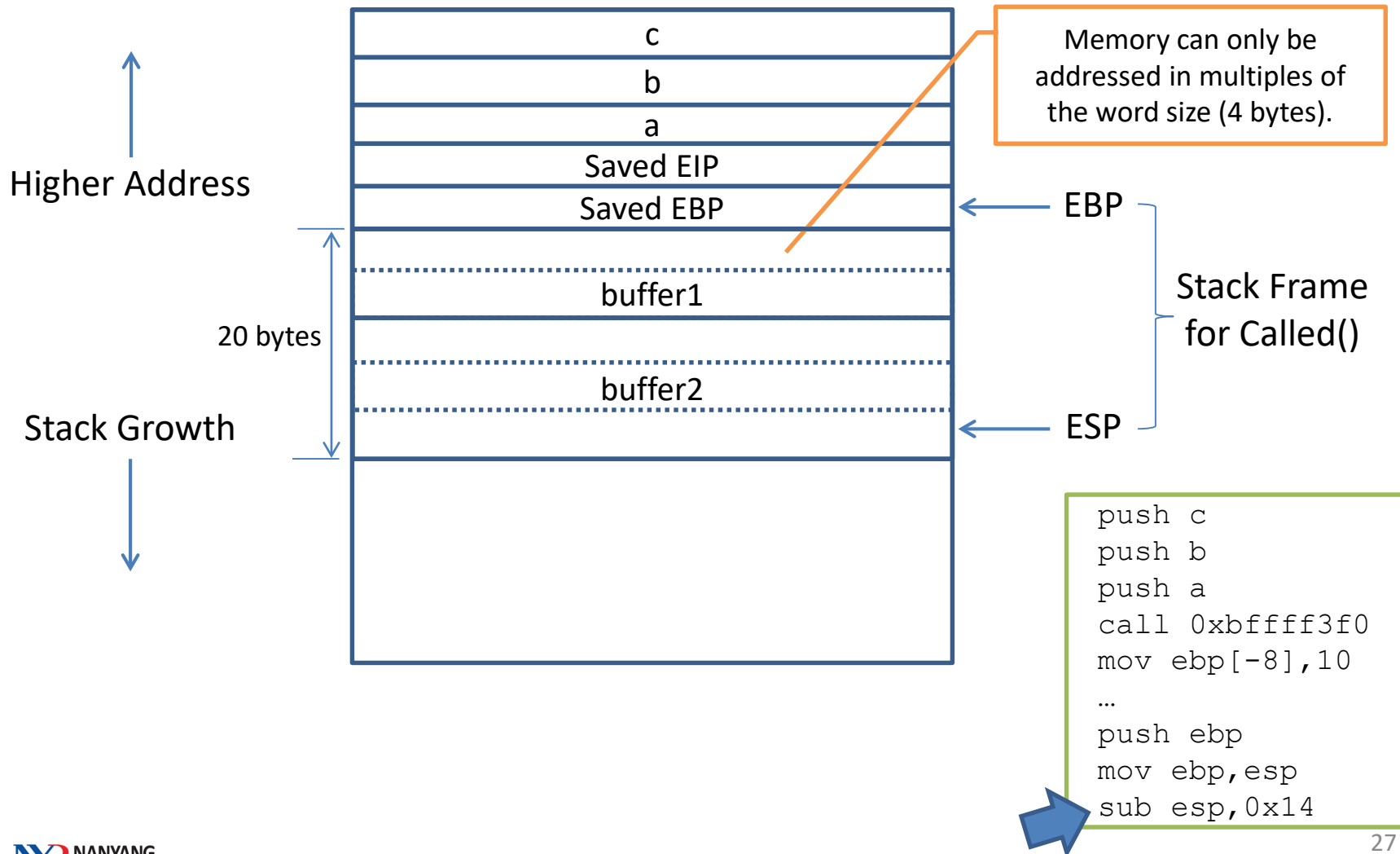
Function Call



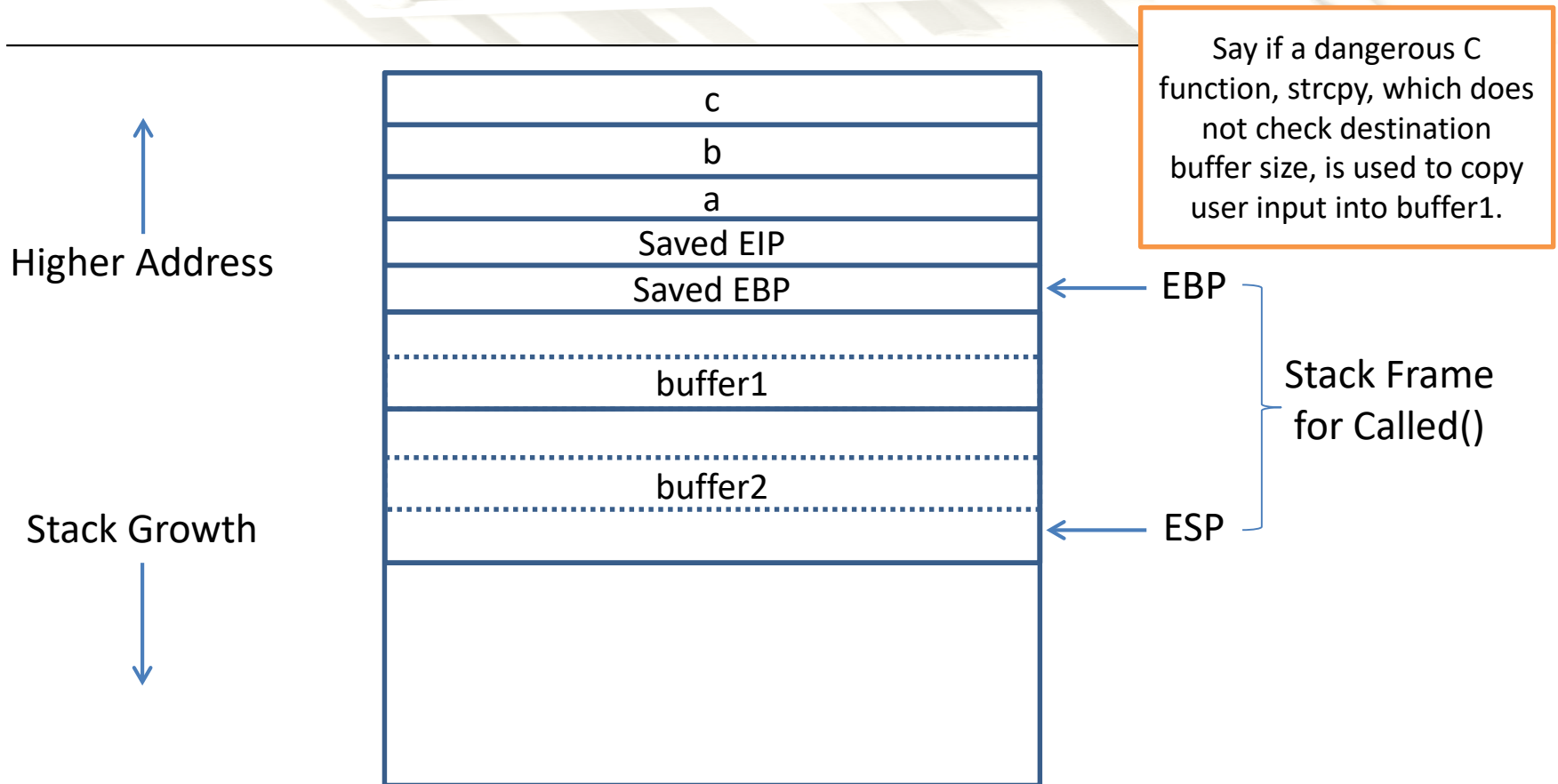
Function Call



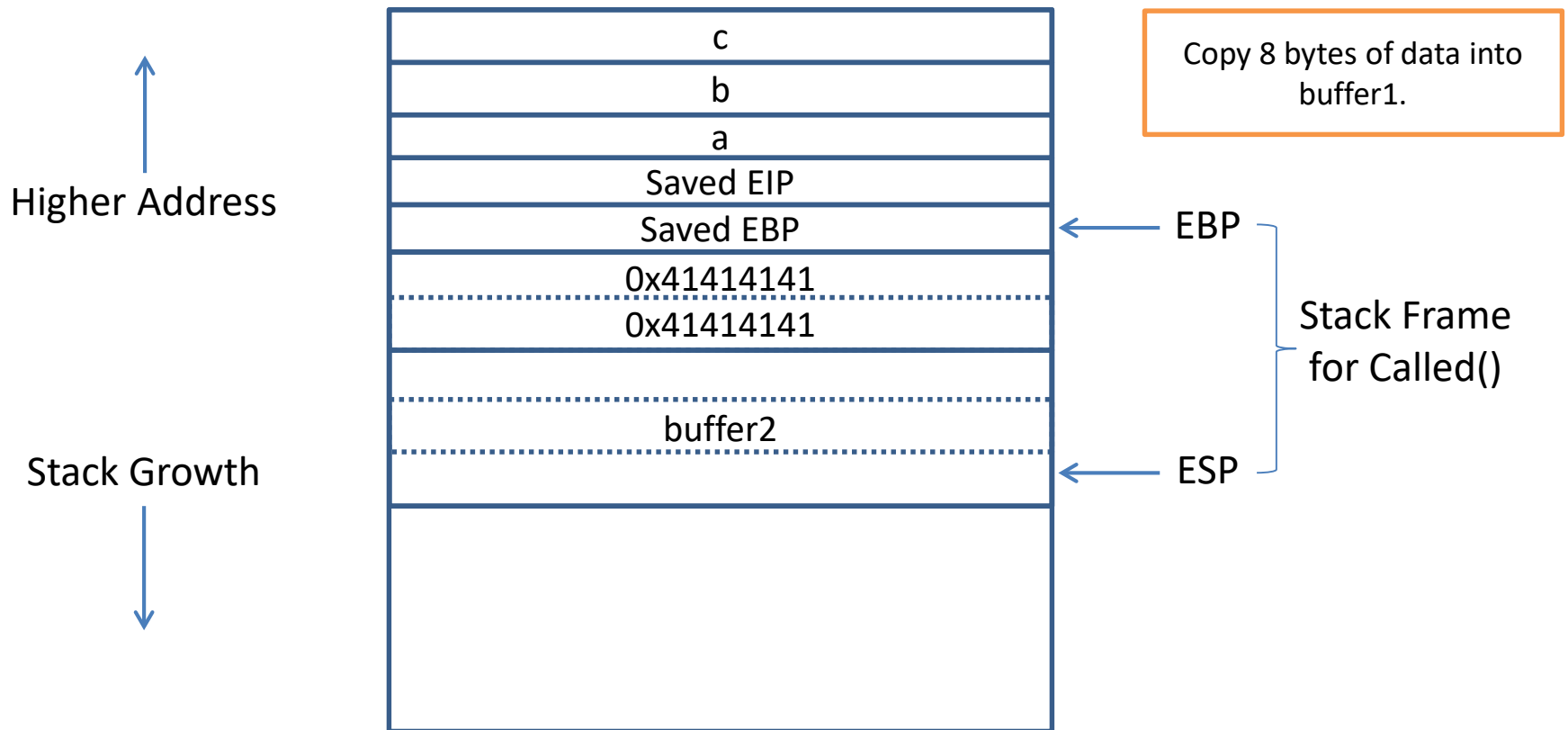
Function Call



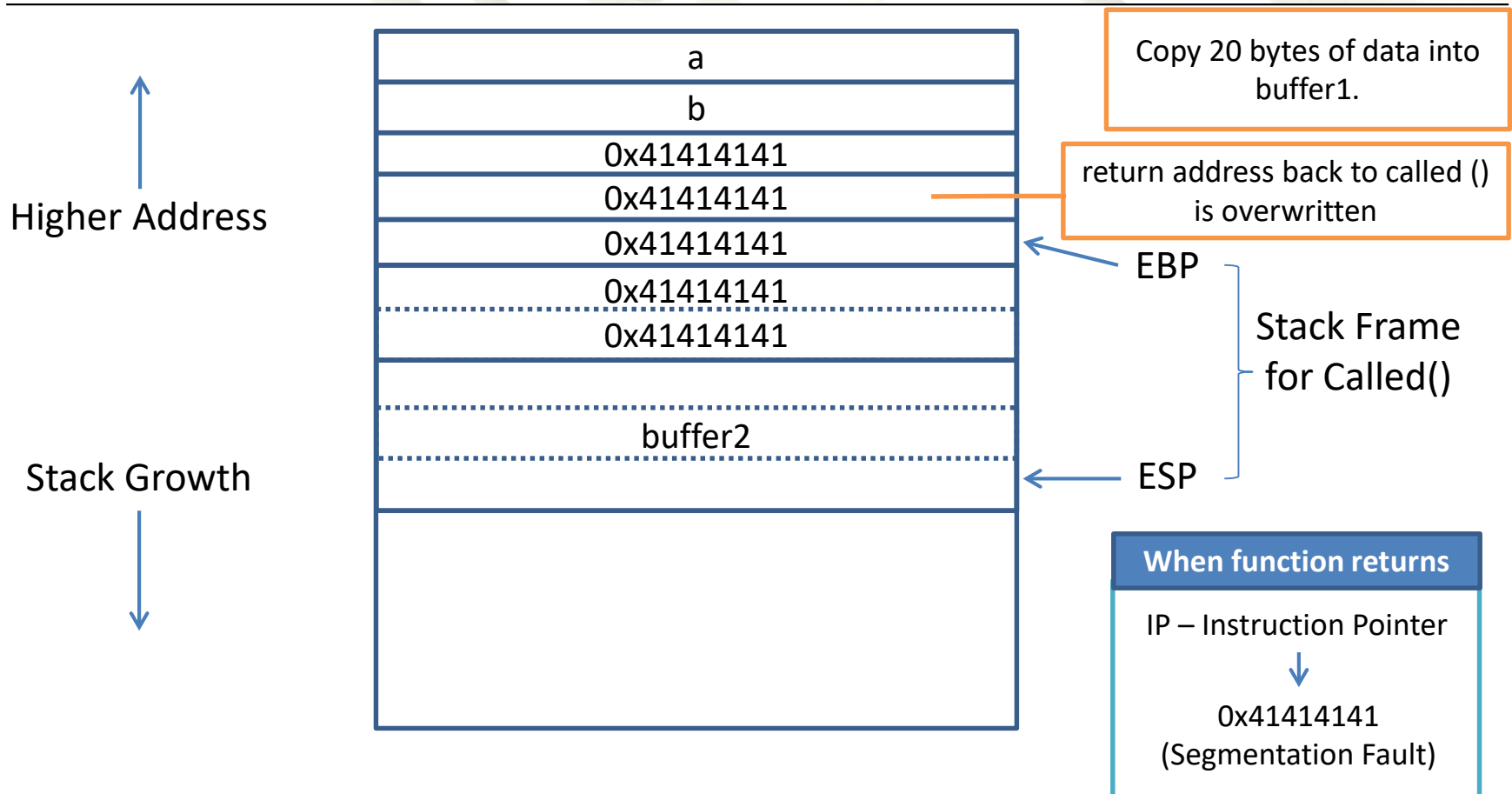
Function Call



Function Call



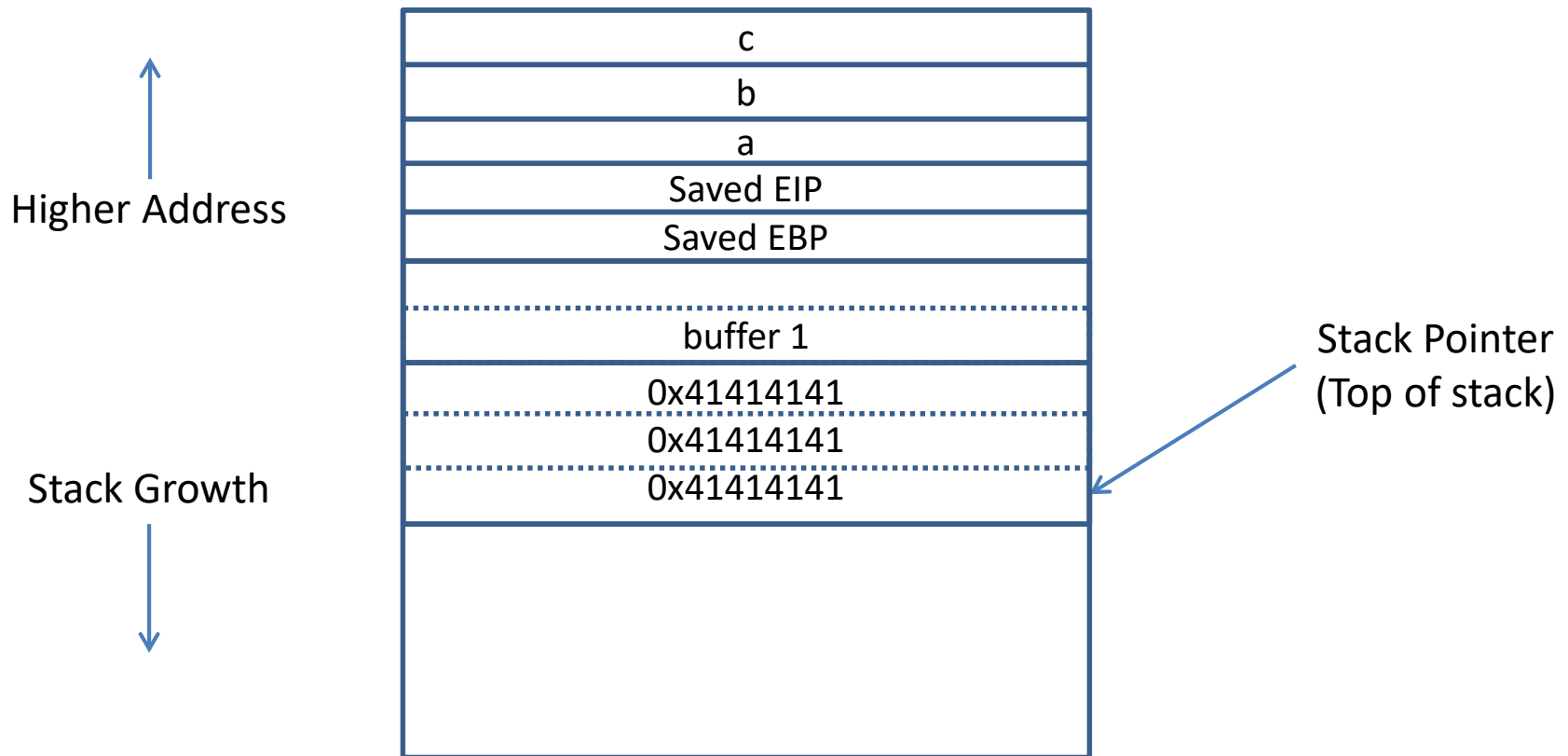
Stack Overflow



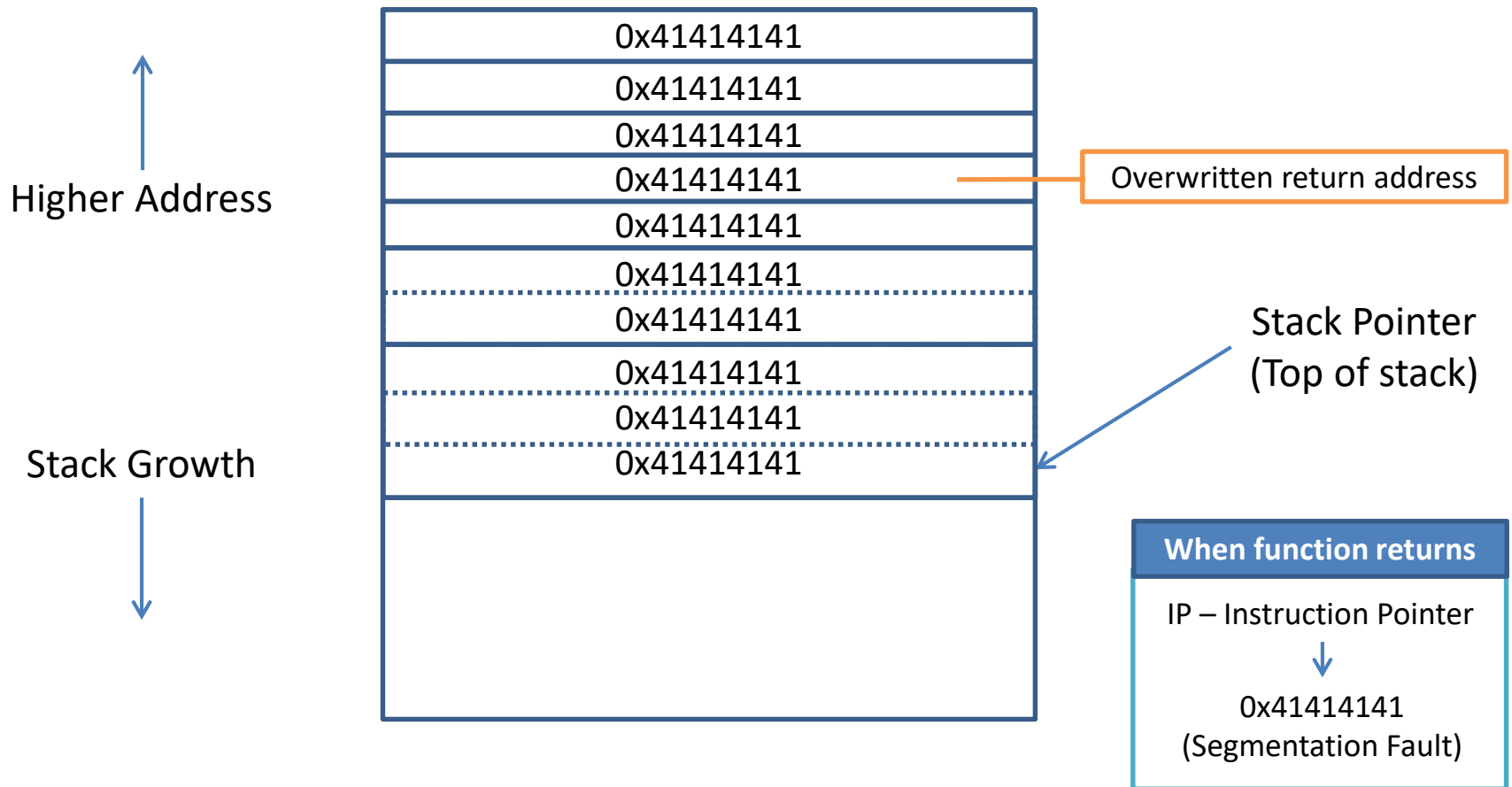
Exploiting Stack Overflow

- 8 bytes above the buffer is the return address to the caller (saved EIP).
 - Saved EIP is 12 bytes from buffer.
 - Next 4 bytes will cause saved EIP to be overwritten.
- Objectives
 - Modify return address to caller so that value of EIP can be controlled.
 - Change the flow of the execution to execute shellcode and maybe spawn a shell.

Function Call



Stack Overflow



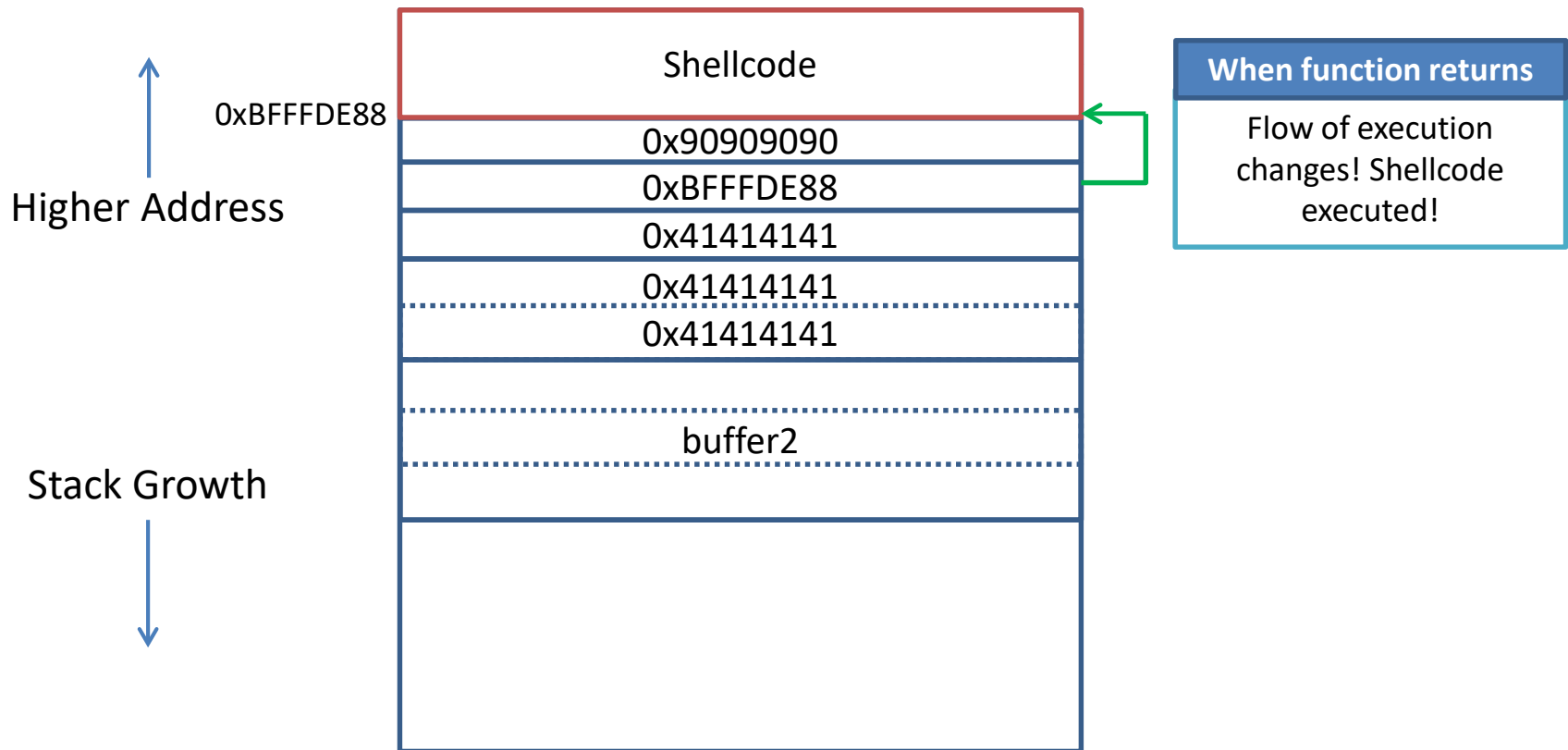
Exploiting Stack Overflow

- 8 bytes above the buffer is the return address to the caller (saved EIP).
 - Saved EIP is 12 bytes from buffer.
 - Next 4 bytes will cause saved EIP to be overwritten.
- Objectives
 - Modify return address to caller so that value of EIP can be controlled.
 - Change the flow of the execution to execute shellcode and maybe spawn a shell.

Steps to Stack Overflow Exploitation

1. Overwrite the buffer with a input of > 16 bytes.
2. Modify return address to caller (saved EIP) to the address where the shellcode (payload) resides.
 - Saved EIP is located at 13th to 16th byte in the buffer.
3. Inject payload after saved EIP.
4. When function returns, the execution flow of the program changes to execute payload.
 - Saved EIP will be restored to the EIP register.
 - Depend on what type of payload, a shell maybe spawned.

Exploiting Stack Overflow

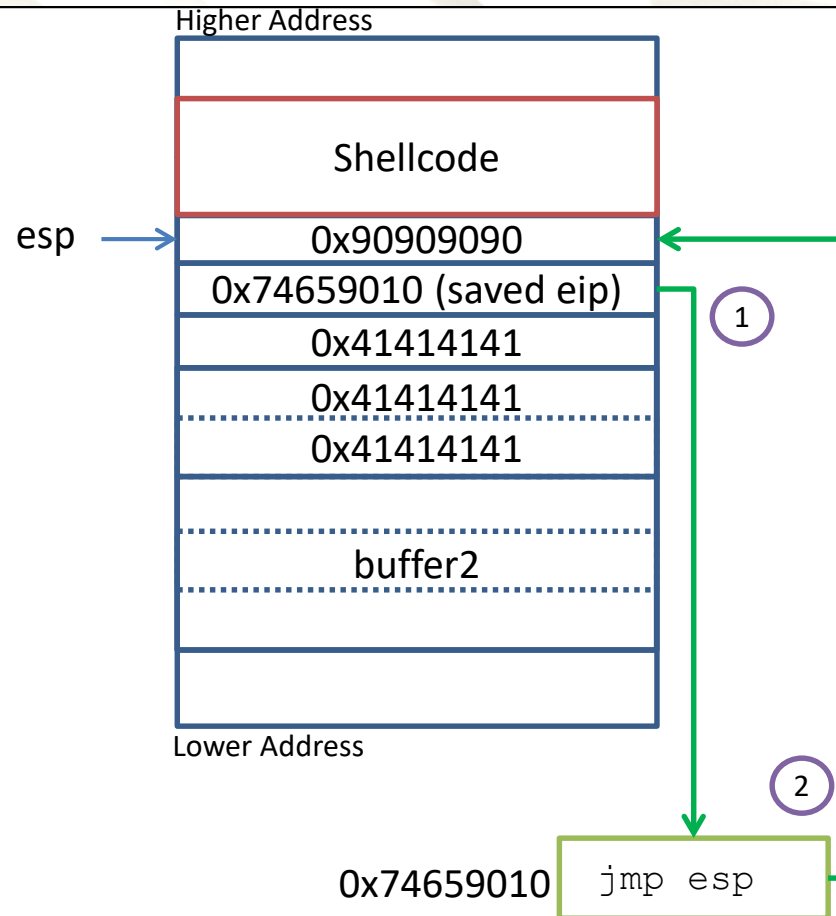


Jumping to Shellcode

- Jumping directly to a memory address may not be reliable.
 - Address may contain null byte.
 - E.g. 0x10007004
 - Memory address layout may be different based on different situations.
- Use techniques to jump to shellcode more reliably.
 - Ideally to use register or offset to a register as reference.

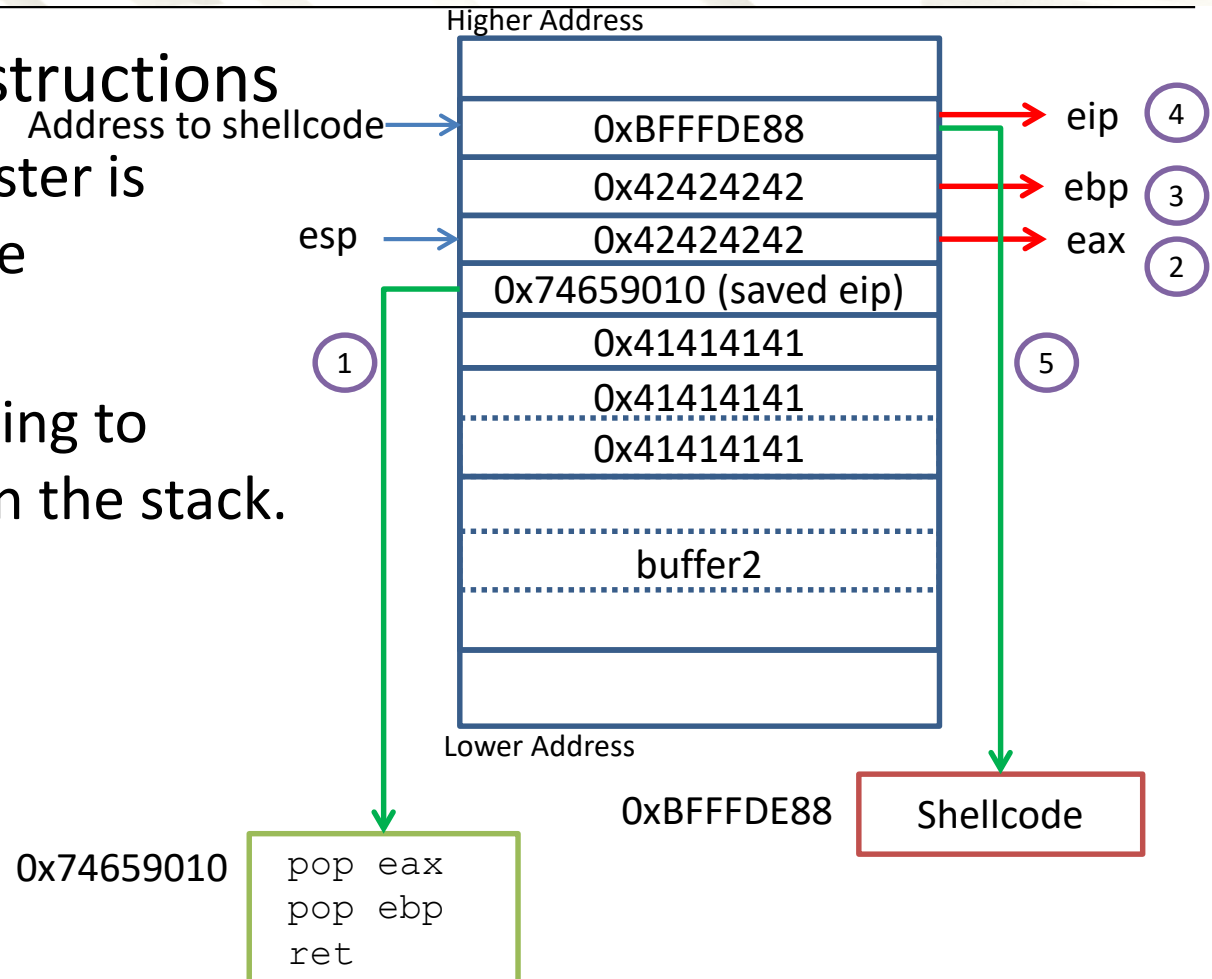
Techniques for Shellcode Execution in Stack Overflow

- Use *jmp/call* to a register
 - When value in one of the registers points to the shellcode.
 - Example
 - Use `jmp esp` if payload is found at the top of the stack.



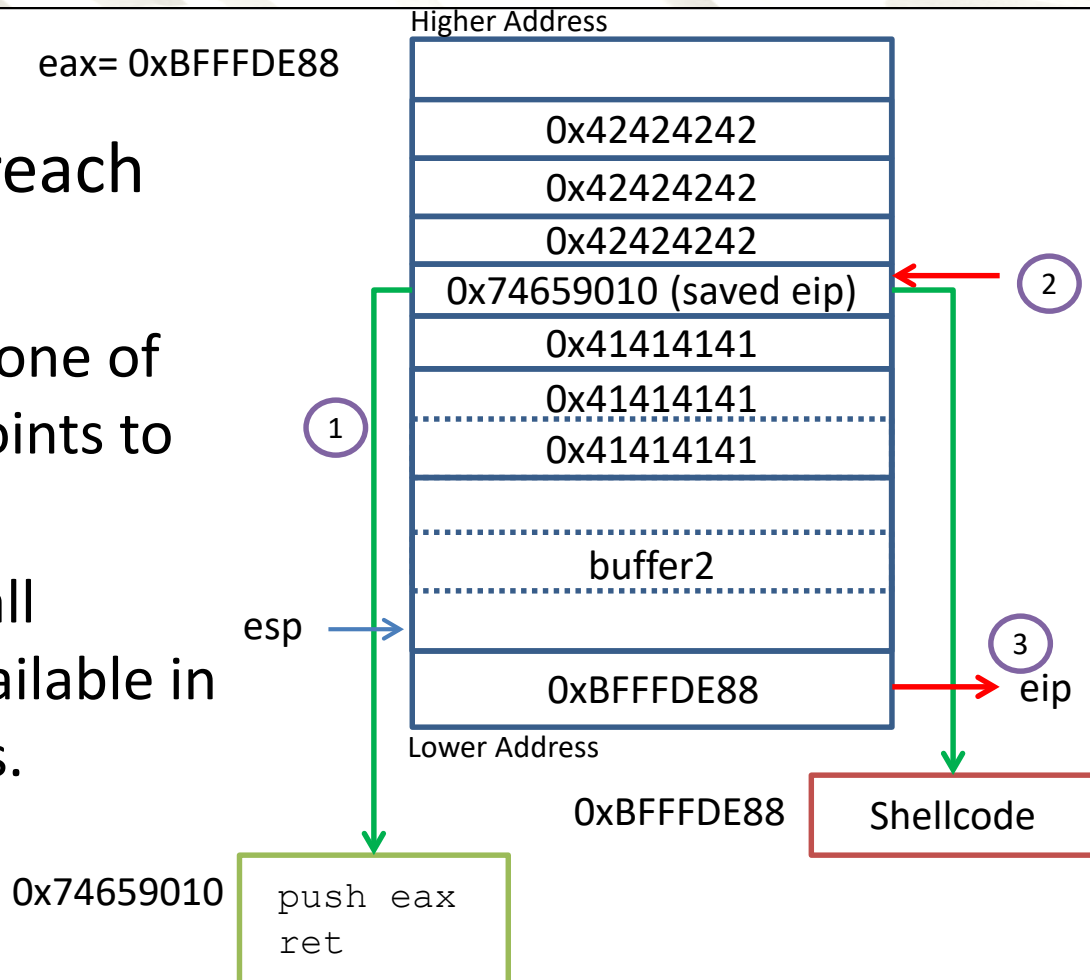
Techniques for Shellcode Execution in Stack Overflow

- Use *pop ret* instructions
 - When no register is pointing to the shellcode.
 - Address pointing to shellcode is on the stack.



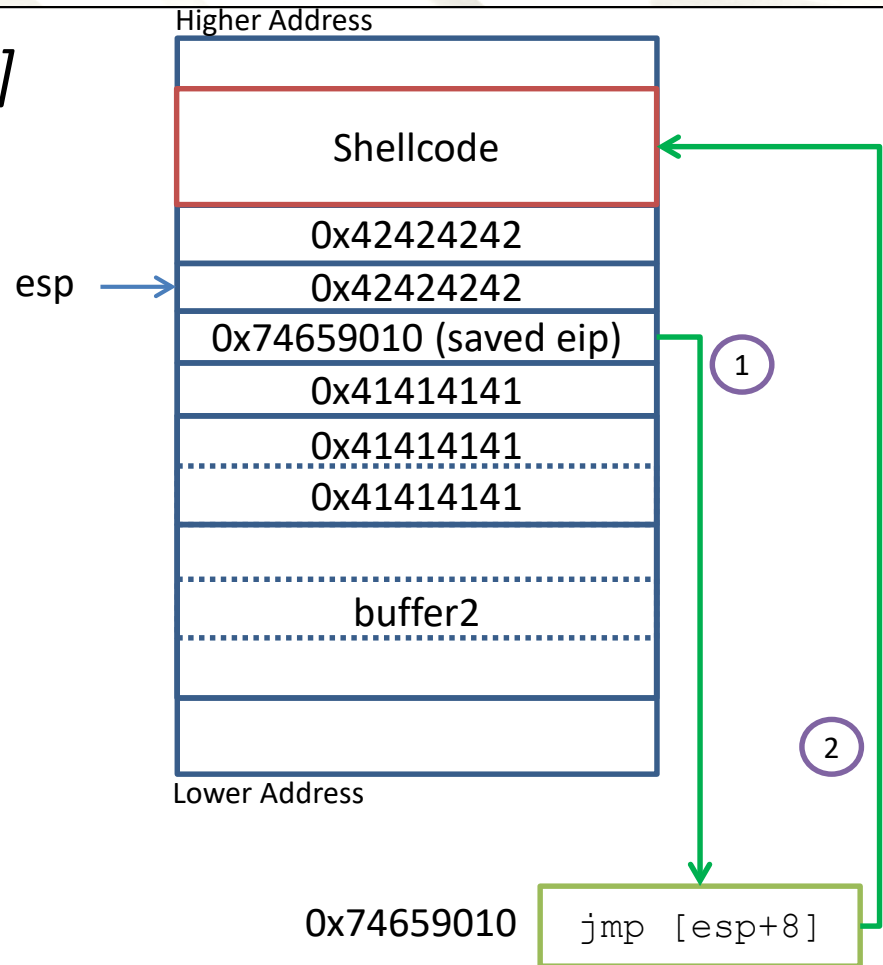
Techniques for Shellcode Execution in Stack Overflow

- Use *push ret* instructions to reach shellcode.
 - When value in one of the registers points to the shellcode.
 - But no jump/call instructions available in loaded libraries.



Techniques for Shellcode Execution in Stack Overflow

- Use *jmp [register + offset]*
 - When value in one of the registers points to somewhere near the shellcode but not at the start of the shellcode.



Buffer Overflow Exploitation (1)

Win32 Assembly

- Application Memory Layout
- Process Memory Organization

Fuzzing

Buffer Overflows

- Types of Buffer Overflows

Stack Exploitation

- Important Registers for Stack Operation
- Function Call
- Stack Overflow
- Exploiting Stack Overflow
- Techniques for Shellcode Execution in Stack Overflow