LAB #2
BOMB LAB + PROJECT

System Programming

May 16th, 2019

Dongmin Jo

MOTIVATION

- You will be given some executable files
- We want you to enhance your analytical skills by debugging them
  - Linux => Bomb Lab
  - Windows => Project

EXECUTABLE FILE

- A file can be executed so that it can be a “process” on memory
- File Format
  - Linux => ELF(Executable and Linkable Format)
  - Windows => PE(Portable Executable) format (exe, dll, sys)

```
dmj@ubuntu:~/Desktop/prac$ xxd -l 16 ./a.out
00000000: 7f45 4c46 0201 0103 0000 0000 0000 0000  .ELF............
```

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 05 00  ........PE..L...
```

BOMB LAB

BOMB LAB OVERVIEW

- You will be given a Linux Executable File (ELF File)
  - Named “bomb”
  - Depending on your student number, you will be given different bombs
- The file has phases which you must defuse with correct inputs
- Whenever you clear the phases one by one, you will get points
- Cautions!
  - If your inputs are incorrect, the bomb will explode
  - Whenever the bomb explodes, **you will lose your point**

Incorrect Inputs

DOWNLOAD BOMB EXAMPLE

- Executes Linux commands
  - wget http://csapp.cs.cmu.edu/3e/bomb.tar
  - tar –xf ./bomb.tar
  - cd bomb
  - ls
- You can see how the bomb works
  - cat bomb.c
  - (or) vi bomb.c

```
dmj@ubuntu:~/Desktop/prac/bomb$ wget http://csapp.cs.cmu.edu/3e/bomb.tar
dmj@ubuntu:~/Desktop/prac/bomb$ tar -xf ./bomb.c
```

BOMB SOURCE CODE (BOMB.C)

```c
FILE *infile;

int main(int argc, char *argv[])
{
    char *input;

    if (argc == 1) {
        infile = stdin;
    }

    else if (argc == 2) {
        if (!(infile = fopen(argv[1], "r"))) {
            printf("%s: Error: Couldn't open %s\n", argv[0], argv[1]);
            exit(8);
        }
    }

    /* You can't call the bomb with more than 1 command line argument. */
    else {
        printf("Usage: %s [<input_file>]\n", argv[0]);
        exit(8);
    }
```

BOMB SOURCE CODE (BOMB.C)

```
dmj@ubuntu:~/Desktop/prac/bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
"HEELLOO"

BOOM!!!
The bomb has blown up.
```

or with file

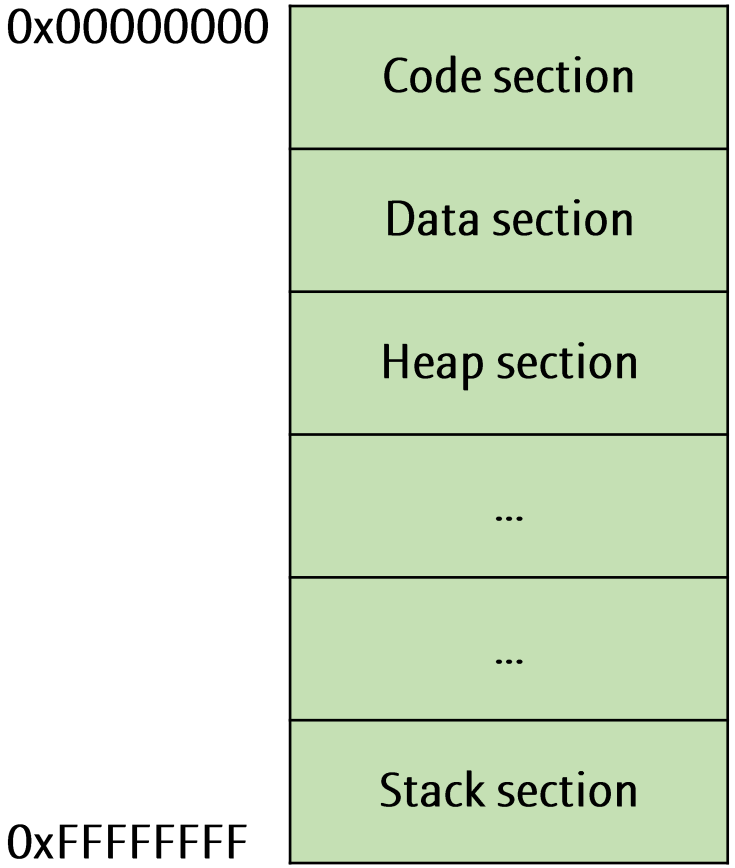
```
dmj@ubuntu:~/Desktop/prac/bomb$ cat hello
phase1 input
phase2 input
and so on
dmj@ubuntu:~/Desktop/prac/bomb$ ./bomb hello
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!

BOOM!!!
The bomb has blown up.
```

BOMB SOURCE CODE (BOMB.C)

```
    printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
    printf("which to blow yourself up. Have a nice day!\n");

    /* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                                      * Let me know how they did it. */
    printf("Phase 1 defused. How about the next one?\n");

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();
    printf("That's number 2.  Keep going!\n");

```

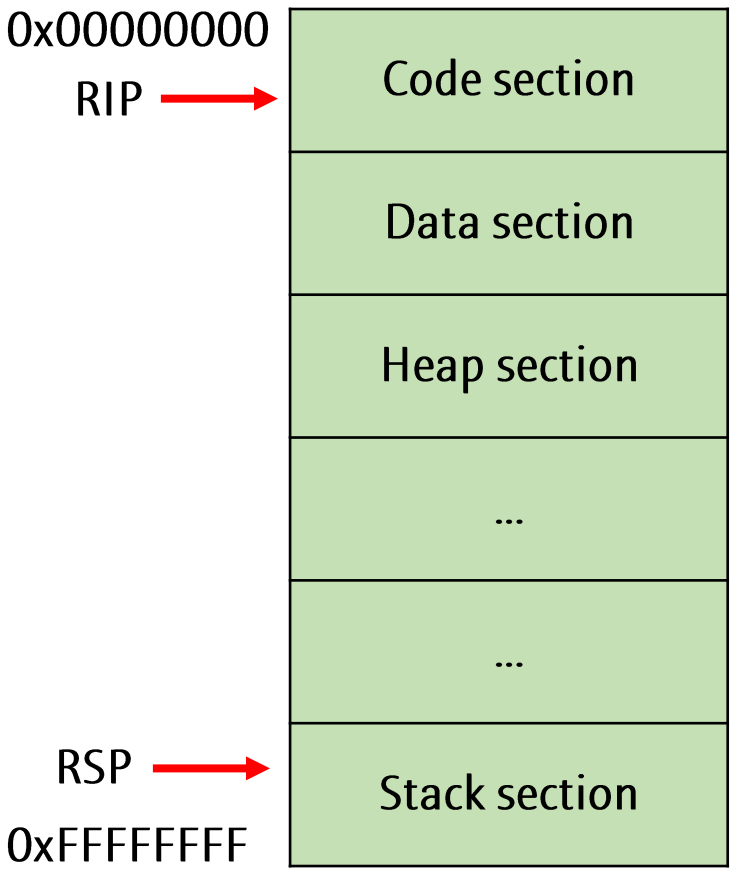
MEMORY STRUCTURE REVIEW

- The executable file is allocated on memory when executed
- Code section
  - Your code as assemblies
- Data section
  - Global variables, constant strings (e.g. “Hello World”)
- Heap section
  - Dynamic allocation such as “malloc” function
- Stack section
  - Local variables



REGISTERS REVIEW

- All registers can store any variables but…
- RIP
  - A values pointing current execution
- RSP
  - A values pointing current stack
- RAX
  - A return value as soon as process exits a function
- RDI, RSI, RDX, RCX, r8, r9
  - 1,2,3,4,5,6th arguments
  - Arguments after 7th are stored in the stack section



LET'S DEFUSE IT

- Use GDB for defusing it

```
dmj@ubuntu:~/Desktop/prac/bomb$ gdb -q ./bomb
Reading symbols from ./bomb...done.
```

DEMO

GDB COMMANDS

- disas [function name]
  - Disassemble the function
  - E.g.) disas main
- b [function name] or [address]
  - Breakpoint
  - The program stops right before the function starts
    - when $RIP == “function start address”
- r (run)
  - Execute the program

GDB COMMANDS

- r (run)
  - Start the program
- si (step instruction)
  - Execute one instruction
  - If it executes “call func”, it steps into the function
- ni (next instruction)
  - Execute one instruction
  - If it executes “call func”, it steps over the function
- c (continue)
  - Execute the program until it stops on breakpoint

GDB COMMANDS

- i r (info registers)
  - Show registers
  - E.g.) i r $rsp
- x/[format] [address]
  - Show data on address
  - E.g.) x/x 0x12345678
    - Show 4 bytes on 12345678 with hexadecimal format
  - E.g.) x/10x 0x12345678
    - Show 4*10 bytes on 12345678 with hexadecimal format
  - E.g.) x/s 0x12345678
    - Show a string on 12345678

HOW TO DEFUSE

- Analyze the phase functions
  - disas “phase_n”
- If you want the bomb not to explode, don’t execute the explosion functions
  - Utilize breakpoints, ni, si well
  - Use c(continue) carefully
  - Check the function names carefully
- Check whether your input is correct carefully
  - Utilize si, ni carefully to check whether your program executes the explosion functions

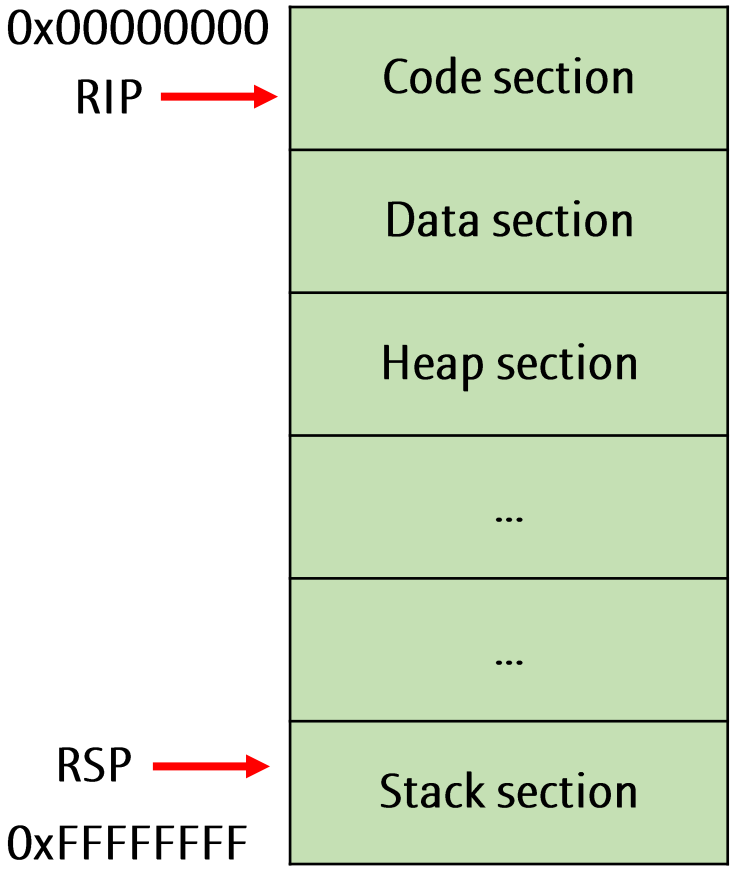
PROJECT

PROJECT OVERVIEW

- You will be given Windows exe files (PE formats)
  - 32 bit files (for x86)
  - Named “crack me”
- You have to find the correct input
- Assemblies will be shown with **“Intel notation”**
  - The source operand and the destination operand are swapped
  - mov a, b => mov b, a
  - e.g.) mov %eax, %esi => mov esi, eax

32 BIT REGISTERS

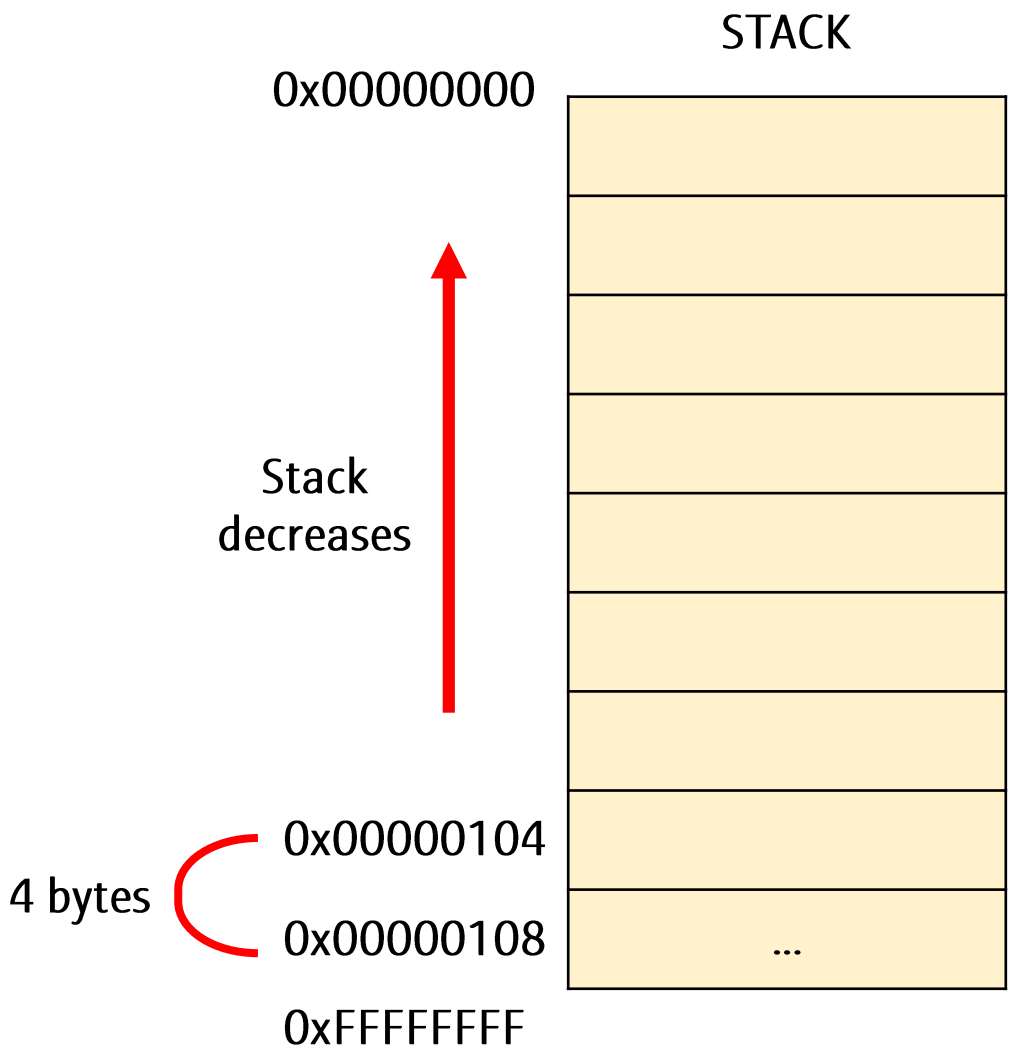
- All registers can store any variables but…
- EIP
  - A values pointing current execution
- ESP
  - A values pointing current stack
- EAX
  - A return value as soon as process exits a function
- Functions arguments goes to stack



PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

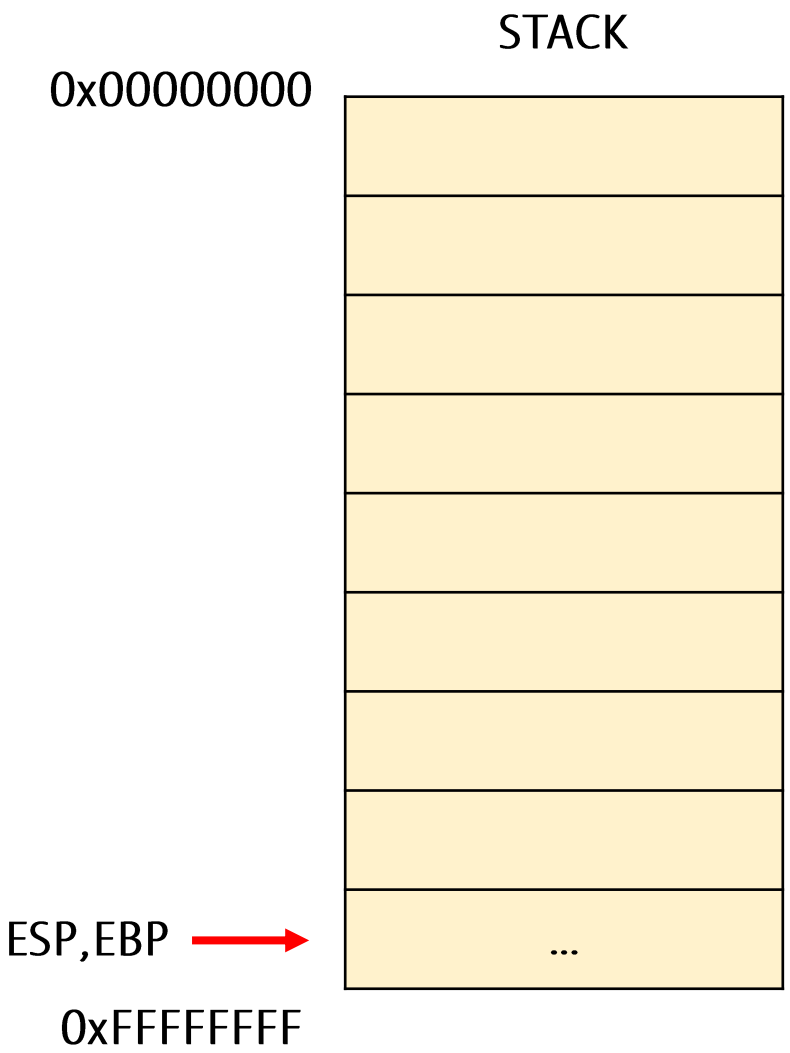
- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret



PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

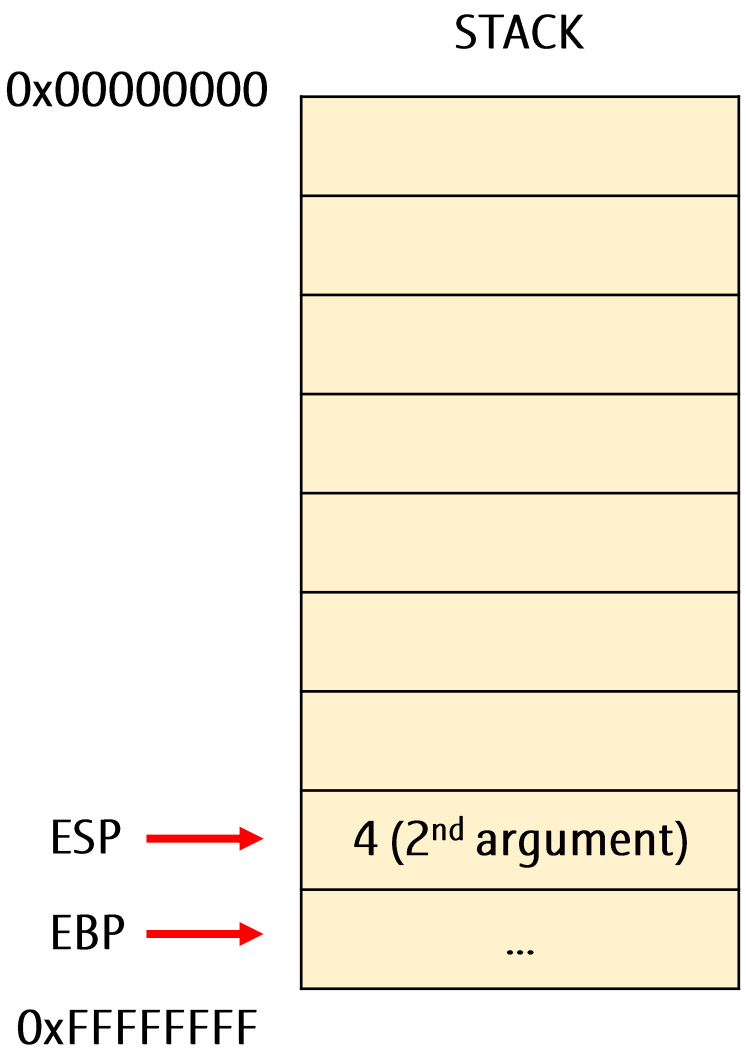
- push 4 ← RIP
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret



PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

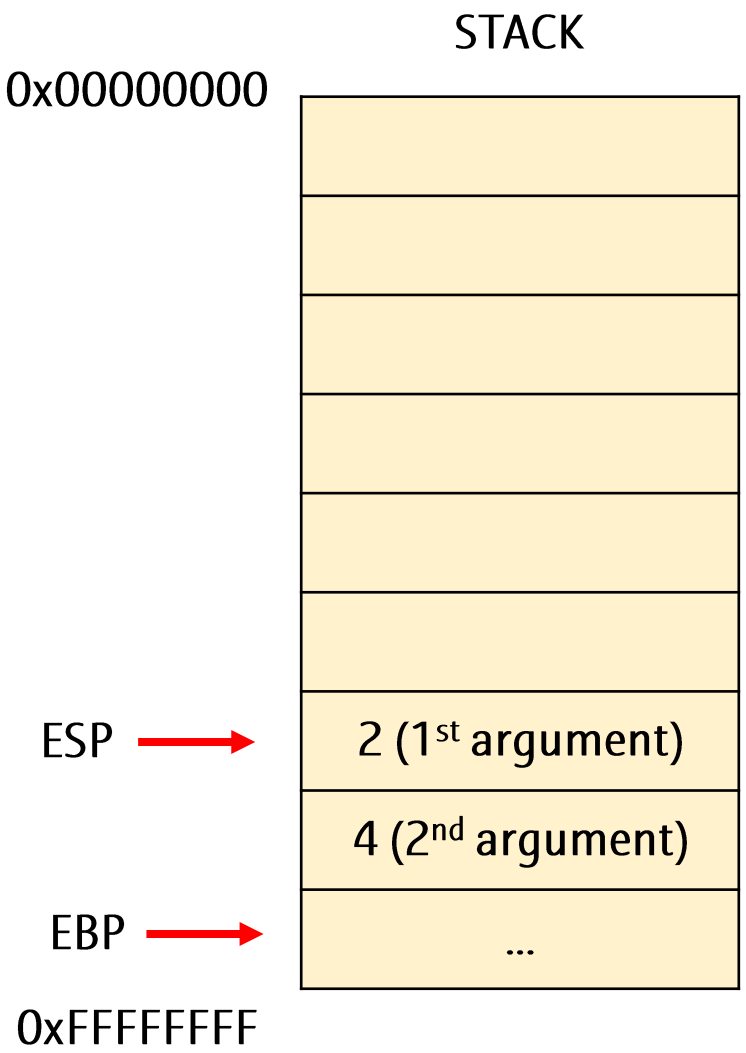
- push 4
- push 2 ← RIP
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret



PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

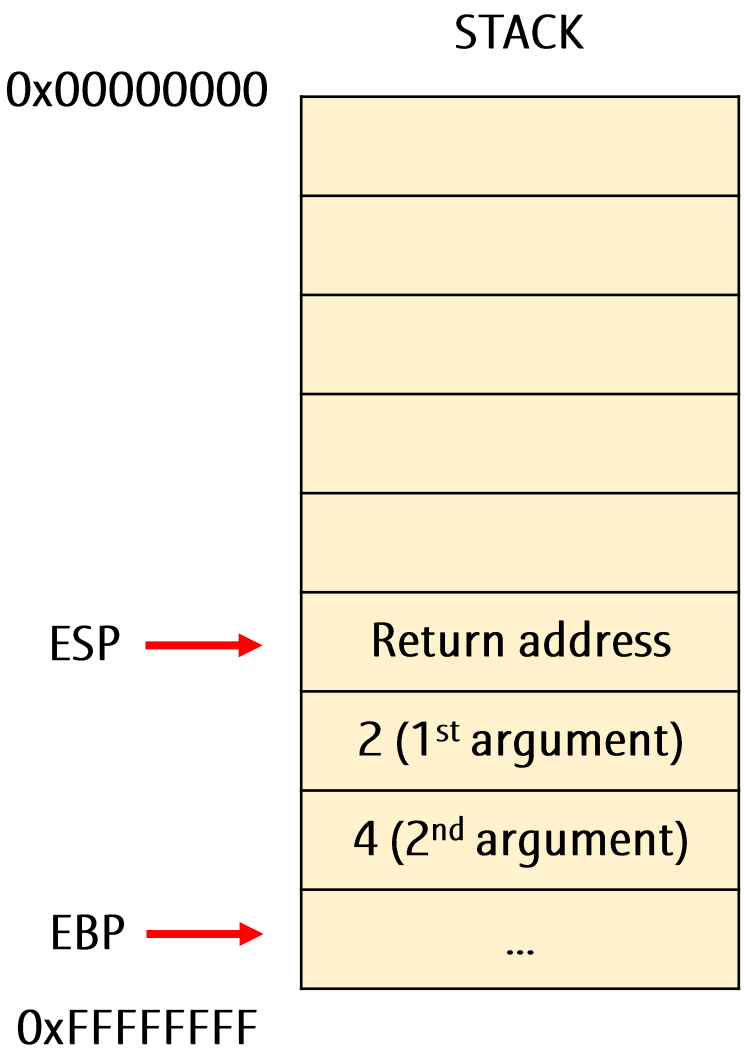
- push 4
- push 2
- call add ← RIP
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret



PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

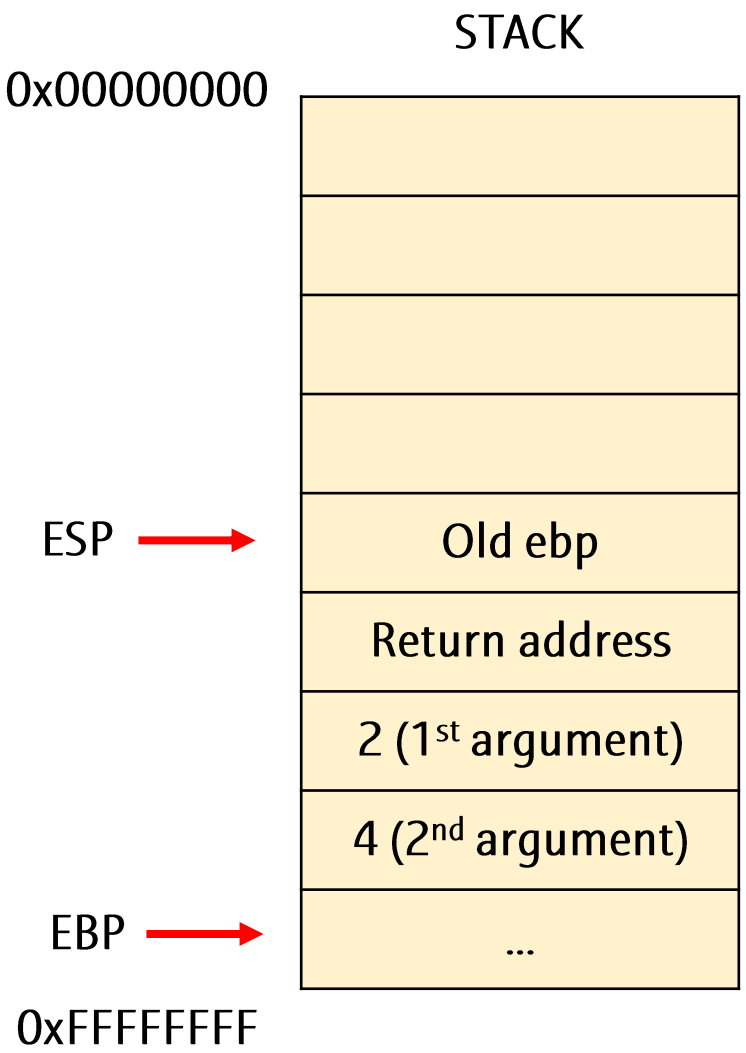
- push 4
- push 2
- call add
  - push ebp ← RIP
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret



PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

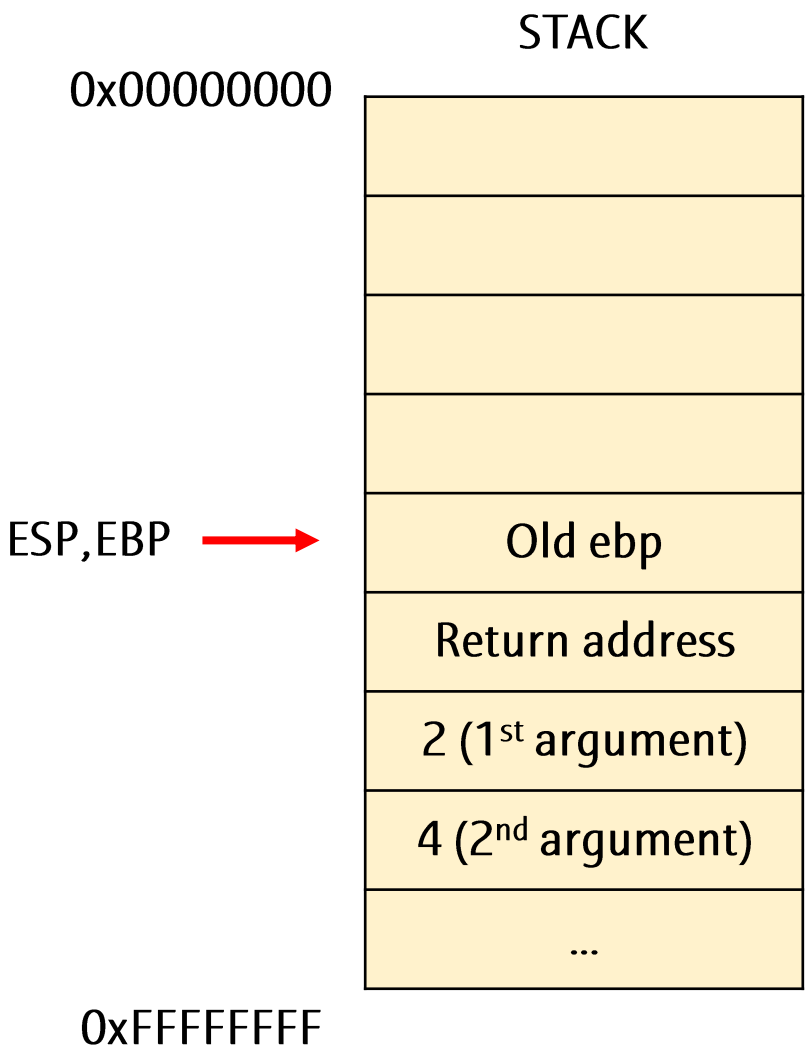
- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp ← RIP
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret



PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8] ← RIP
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret

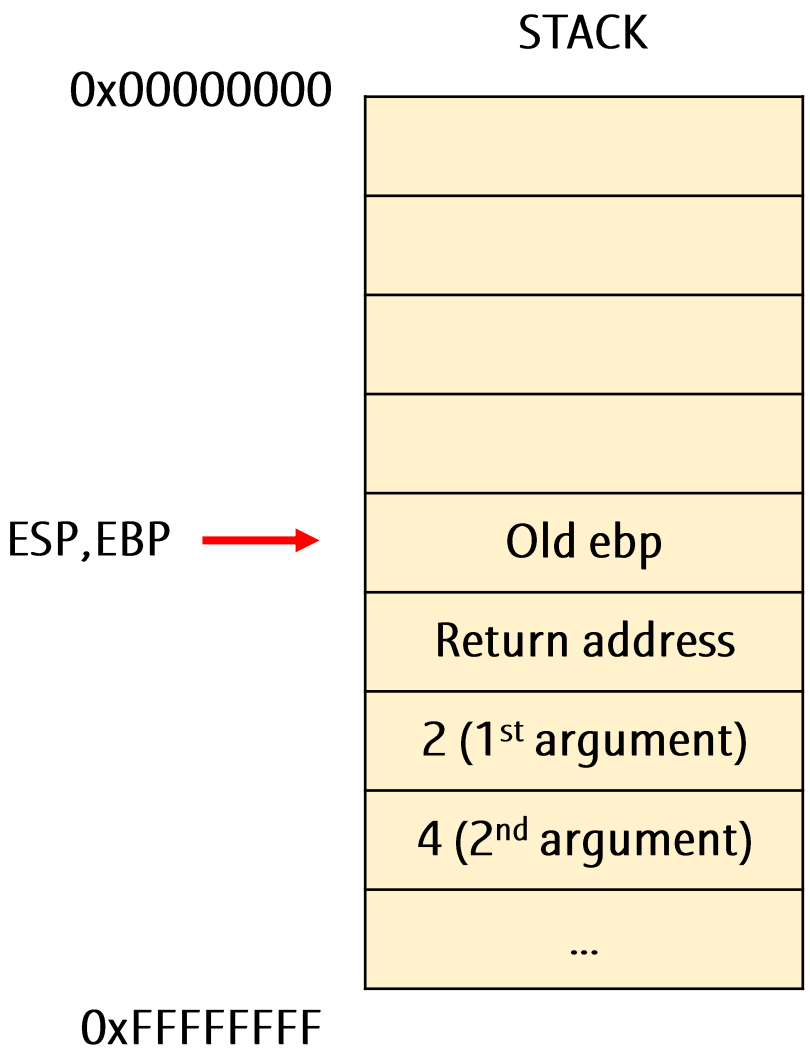


PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12] ← RIP
  - mov esp, ebp
  - pop ebp
  - ret

EAX = 2



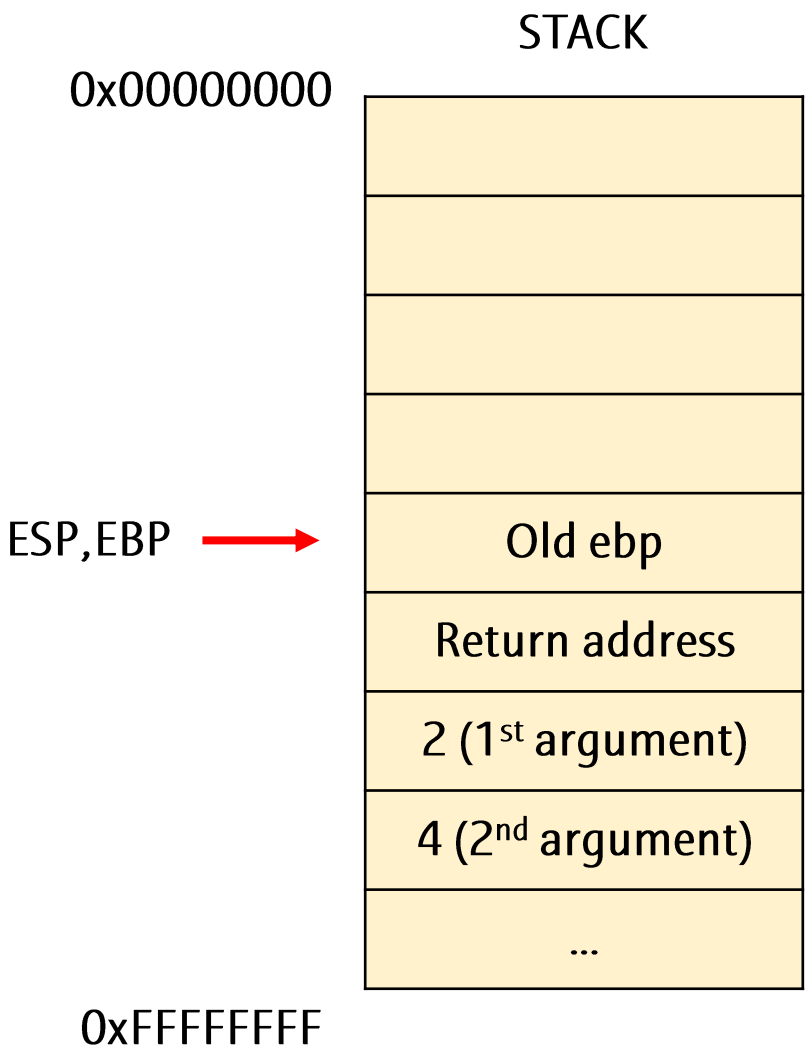
PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp ← RIP
  - pop ebp
  - ret

EAX = 2

EAX = 2 + 4 = 6



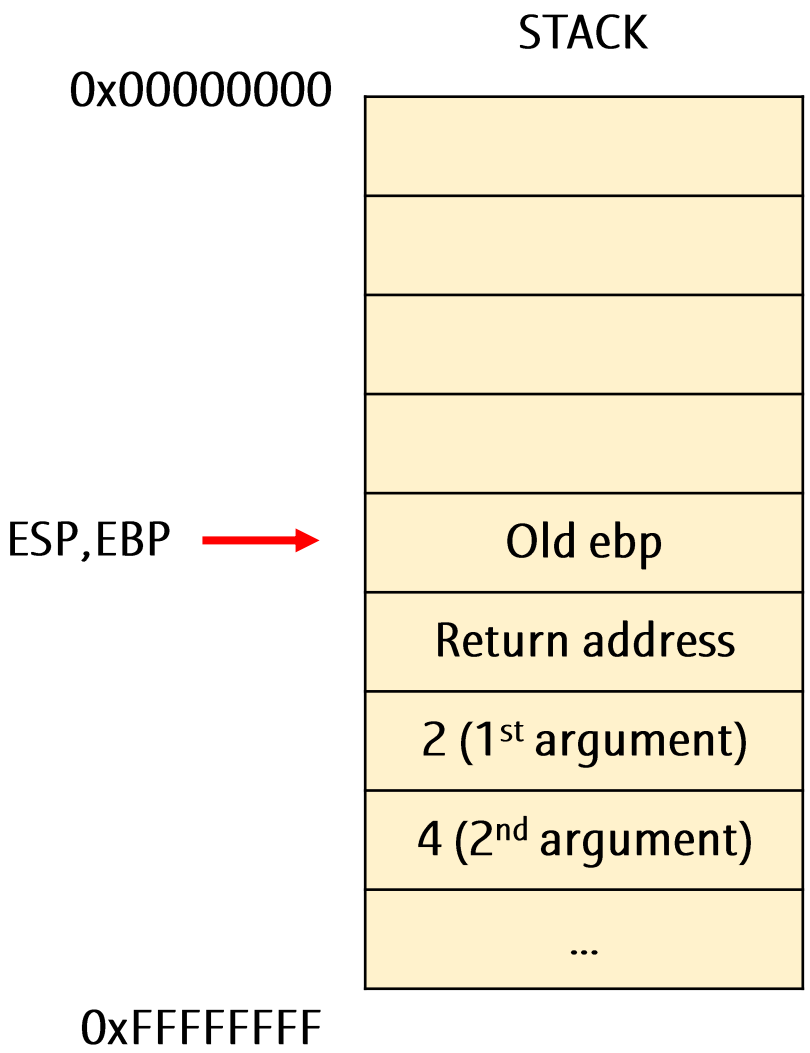
PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp ← RIP
  - ret

EAX = 2

EAX = 2 + 4 = 6



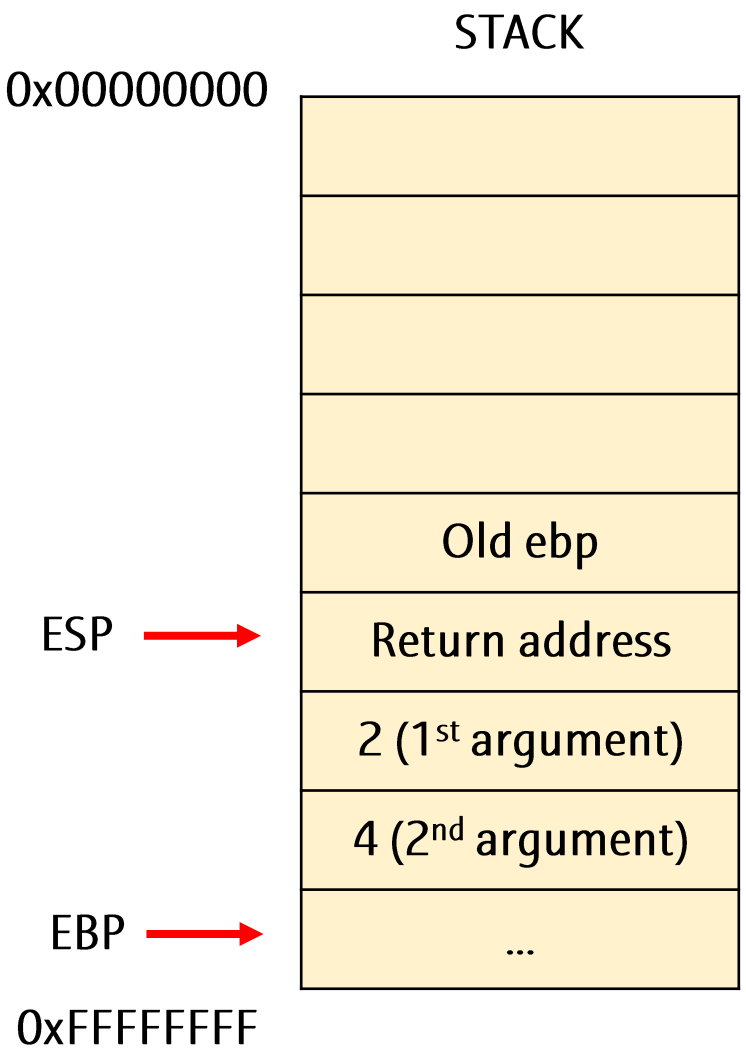
PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret ← RIP

EAX = 2

EAX = 2 + 4 = 6



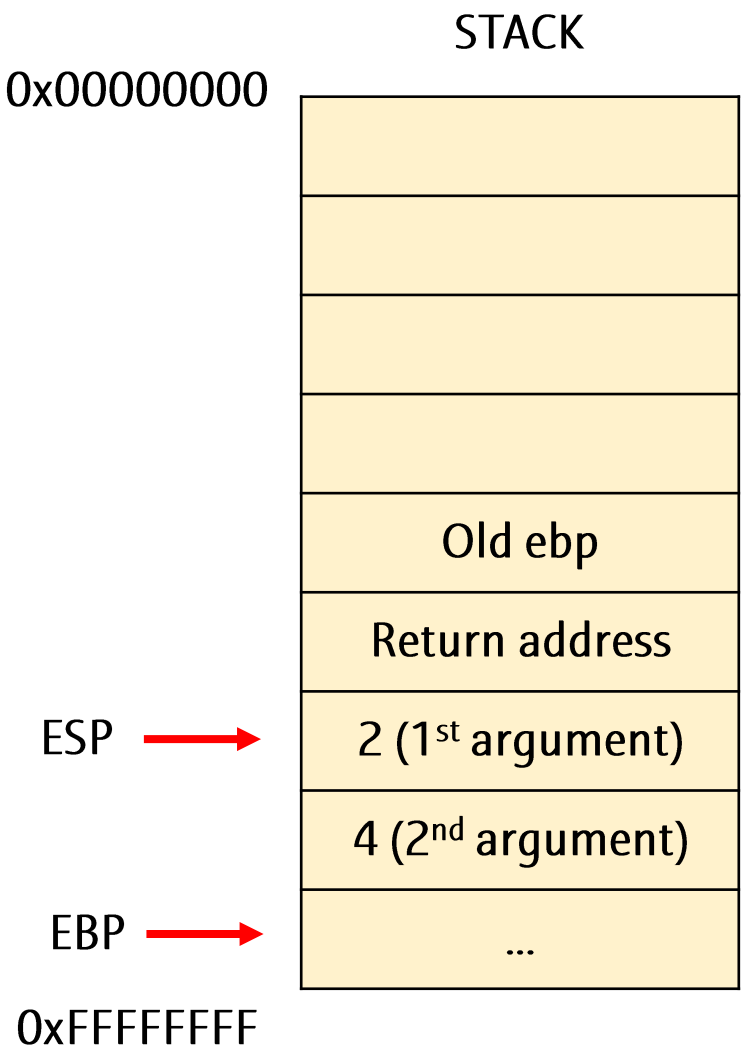
PASSING ARGUMENTS ON X86

```
1 int add(int a, int b){
2     return a + b;
3 }
4 add(2,4);
```

- push 4
- push 2
- call add
  - push ebp
  - mov ebp, esp
  - mov eax, [ebp+8]
  - add eax, [ebp+12]
  - mov esp, ebp
  - pop ebp
  - ret

EAX = 2

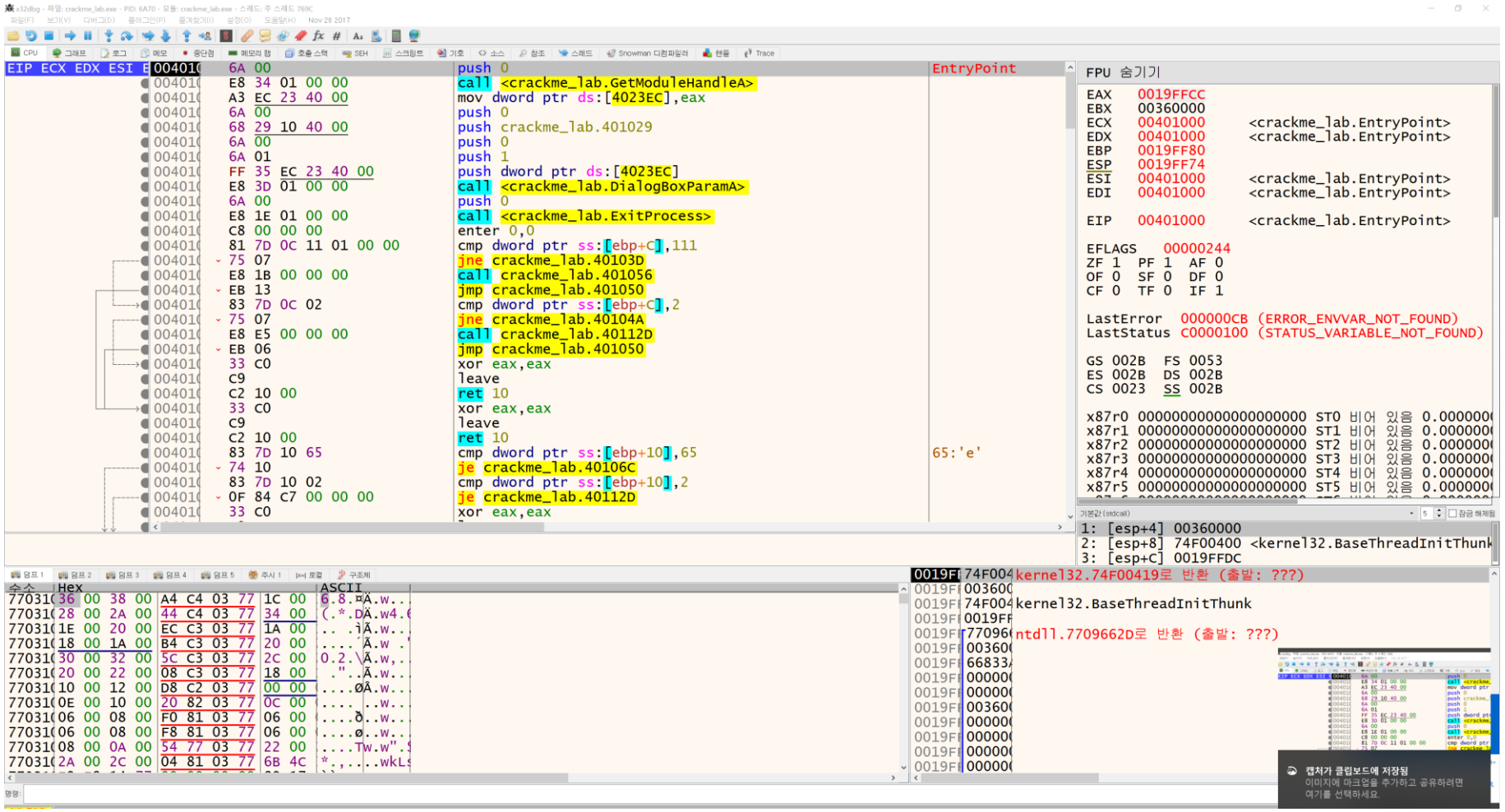
EAX = 2 + 4 = 6



LET'S CRACK IT

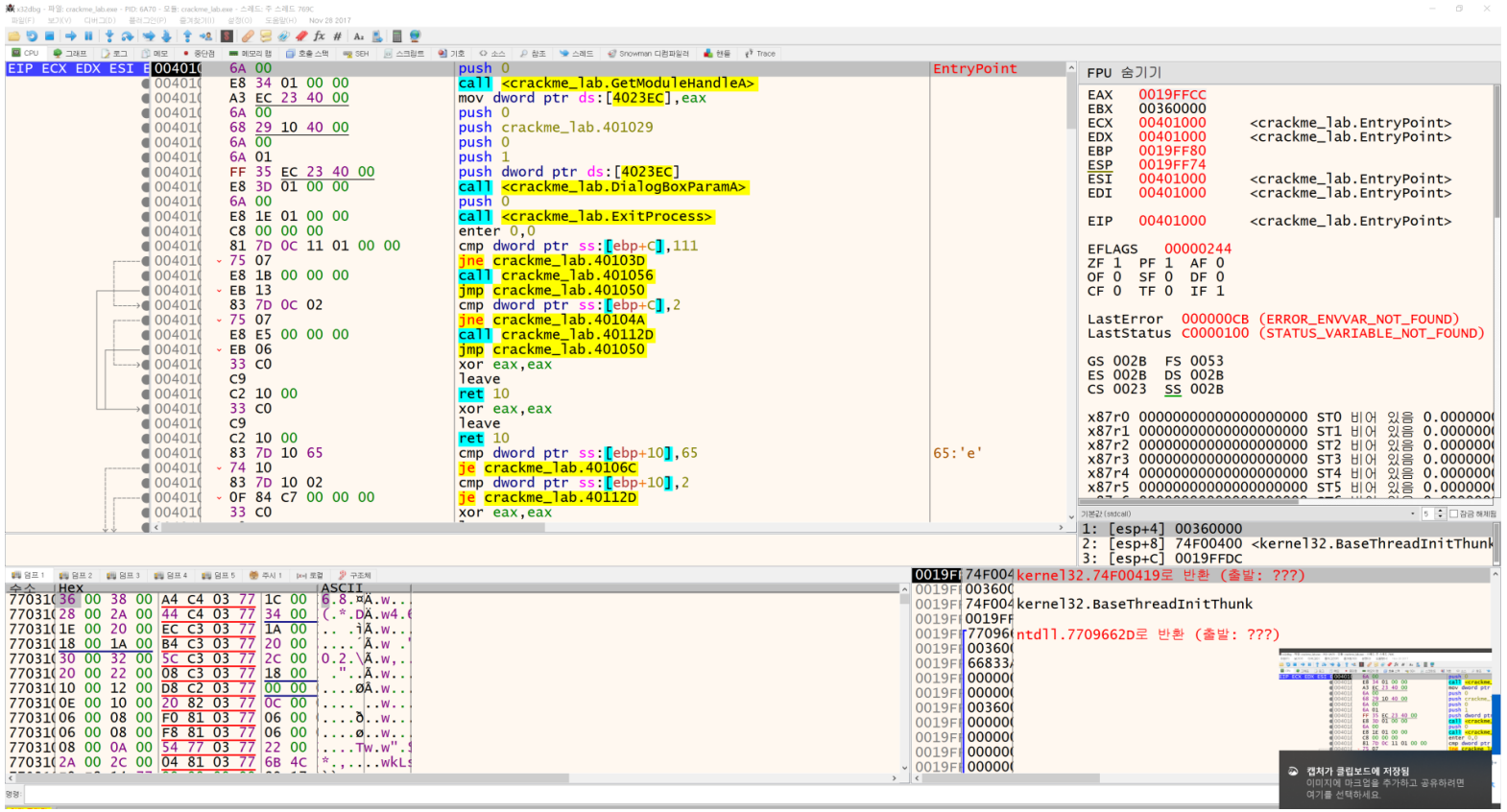
- Download crackme_lab.exe
- Start x32dbg
- Drag crackme_lab.exe into debugger

X32DBG AND X64DBG



X32DBG AND X64DBG

Code section
Data section
Heap section
…
…
Stack section



X32DBG AND X64DBG

DEMO

DEBUGGER COMMANDS

- F2 Key
  - Breakpoint
- F7 Key
  - Step into
- F8 Key
  - Step over
- F9 Key
  - Start or continue the program

WRAP UP

- Google will definitely help you
  - GDB commands, x64 debugger usage
  - Stack frames, calling conventions and so on
- We hope you enjoy the assignments