1 Passing Data to Functions on the Stack

Consider the following assembly program:

```
section .data

section .text

        global main

        addTwo:
                PUSh ebp
                MOV ebp, esp
                ADD eax, ebx

                MOV eax, [ebp+8]
                MOV ebx, [ebp+12]

                ADD eax, ebx
                POP ebp

                RET

        main:
                PUSH 4
                PUSh 1
                CALL addTwo
                MOV ebx, eax

                MOV eax, 1
                INT 80h
```

Here, we are trying to implement parameterized functions!!! To implement parameterized functions in x86, we use the stack(pretty much like for calling standard C functions or even user-defined C functions).

In the program, we are implementing an add function which basically adds two integers and returns their sum.

Snippet from the program:

```
main:
        PUSH 4
        PUSH 1
        CALL addTwo
        MOV ebx, eax

        MOV eax, 1
        INT 80h
```

Here, the first two lines push the values 4 and then 1 into the stack. When we call the **addTwo** function using **CALL** instruction then the return address(the address of instruction after the **CALL** instruction that is **MOV ebx, eax** in our case) is pushed into the stack. So the overall stack looks like this:

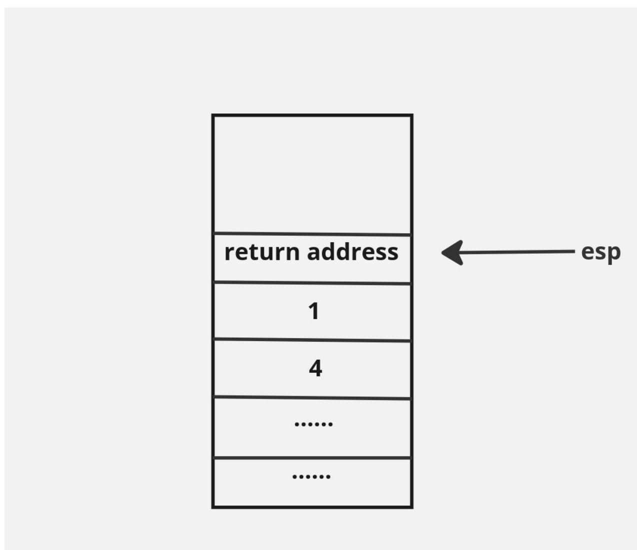


Figure 1: The stack

In order to access the integer values from the stack in **addTwo**, we use this approach:

```
addTwo:
        PUSH ebp
        MOV ebp, esp
```

Here, we are pushing the register **EBP** into the stack. This register basically acts as a base of a stack frame. If we called multiple functions inside a function then we will need something to distinguish between everything that is associated with the first function and everything that is associated with the second function. The **EBP** register is like the divider.

Hence, our stack looks something like this:

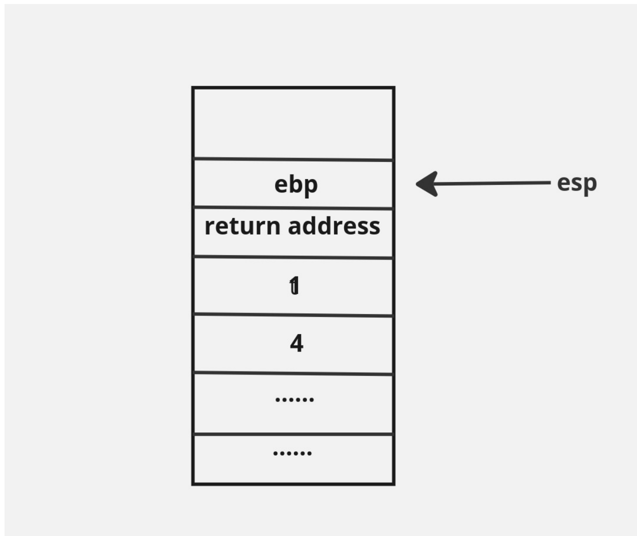


Figure 2: **EBP** inside the stack

We also do **MOV ebp, esp** where we move the value in **ESP** into **EBP** so that **EBP** and **ESP** both the registers are pointing to the same location.

With this, now, we will be referencing values **relative** to **EBP** that is as shown below:

```
addTwo:
        PUSH ebp
        MOV ebp, esp

        MOV eax, [ebp+8]
        MOV ebx, [ebp+12]

        ADD eax, ebx
```

So, here we are accessing the values 1 and 4 from the stack(see figure below), and storing them into registers **EAX** and **EBX** respectively:

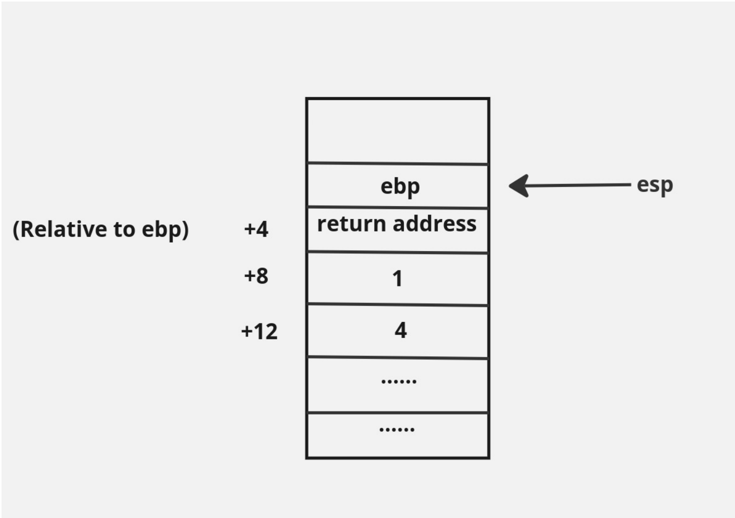


Figure 3: Stack elements relative to **EBP**

So, we can see that the values are 4 bytes apart from each other.

Now, we cannot return from **addTwo** because right now, **ESP** is pointing to **EBP** which is at the top of the stack(see Figure 3). In order to make **ESP** point to the return address in the stack, we will have to **POP EBP** from the stack.

Hence,

```
POP ebp
```

This pops **EBP** from the stack and **ESP** points to the return address which is now the new top of the stack.

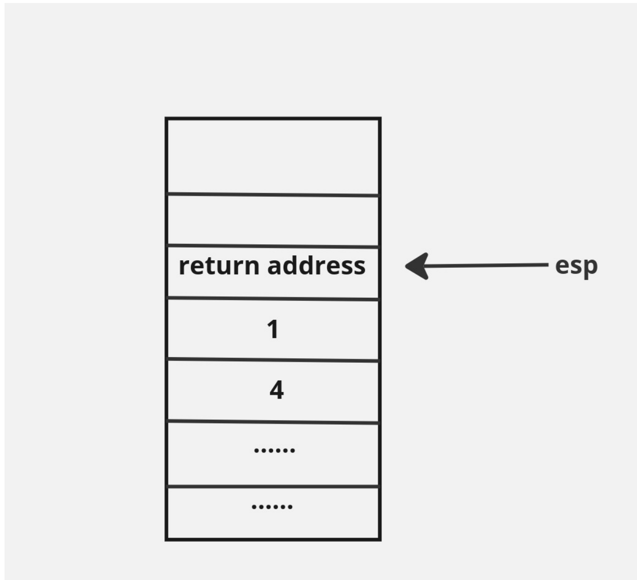


Figure 4: **EBP** popped from stack

Now, we can return back to the caller.

2 Verification

We will use GDB to see how the values get stored into the stack:

```
B+  0x804917d <main>          push   $0x4
    0x804917f <main+2>        push   $0x1
  > 0x8049181 <main+4>        call   0x8049170 <addTwo>
    0x8049186 <main+9>        mov    %eax,%ebx
    0x8049188 <main+11>       mov    $0x1,%eax
    0x804918d <main+16>       int    $0x80
    0x804918f                 add    %dh,%bl
    0x8049191 <_fini+1>       nop    %ebx
    0x8049194 <_fini+4>       push   %ebx
    0x8049195 <_fini+5>       sub    $0x8,%esp
    0x8049198 <_fini+8>       call   0x80490a0 <__x86.get_pc_thunk.bx>
    0x804919d <_fini+13>      add    $0x2e63,%ebx
    0x80491a3 <_fini+19>      add    $0x8,%esp
    0x80491a6 <_fini+22>      pop    %ebx
    0x80491a7 <_fini+23>      ret
    0x80491a8                 add    %al,(%eax)
    0x80491aa                 add    %al,(%eax)
    0x80491ac                 add    %al,(%eax)
    0x80491ae                 add    %al,(%eax)
    0x80491b0                 add    %al,(%eax)
    0x80491b2                 add    %al,(%eax)
    0x80491b4                 add    %al,(%eax)
    0x80491b6                 add    %al,(%eax)
    0x80491b8                 add    %al,(%eax)
    0x80491ba                 add    %al,(%eax)
```

Figure 5: GDB

So, we executed the first two lines.

Once we enter **addTwo**, we will view the value in **ESP**.

```
    0x8049170 <addTwo>        push   %ebp
    0x8049171 <addTwo+1>      mov    %esp,%ebp
  > 0x8049173 <addTwo+3>      mov    0x8(%ebp),%eax
    0x8049176 <addTwo+6>      mov    0xc(%ebp),%ebx
    0x8049179 <addTwo+9>      add    %ebx,%eax
    0x804917b <addTwo+11>     pop    %ebp
    0x804917c <addTwo+12>     ret
B+  0x804917d <main>          push   $0x4
    0x804917f <main+2>        push   $0x1
    0x8049181 <main+4>        call   0x8049170 <addTwo>
    0x8049186 <main+9>        mov    %eax,%ebx
    0x8049188 <main+11>       mov    $0x1,%eax
    0x804918d <main+16>       int    $0x80
    0x804918f                 add    %dh,%bl
    0x8049191 <_fini+1>       nop    %ebx
    0x8049194 <_fini+4>       push   %ebx
    0x8049195 <_fini+5>       sub    $0x8,%esp
    0x8049198 <_fini+8>       call   0x80490a0 <__x86.get_pc_thunk.bx>
    0x804919d <_fini+13>      add    $0x2e63,%ebx
    0x80491a3 <_fini+19>      add    $0x8,%esp
    0x80491a6 <_fini+22>      pop    %ebx
    0x80491a7 <_fini+23>      ret
    0x80491a8                 add    %al,(%eax)
    0x80491aa                 add    %al,(%eax)
```

Figure 6: In **addTwo**

Now, that we are inside **addTwo**, we will check the values of **ESP**, we also pushed **EBP** into the stack and made **EBP** point to the same memory location as **ESP** as shown below:

```
(gdb) info registers esp
esp            0xffffcf3c          0xffffcf3c
(gdb) info registers ebp
ebp            0xffffcf3c          0xffffcf3c
(gdb)
```

Figure 7: Values in **ESP** and **EBP**

Now, we will see what is stored in the address **0xffffcf3c**.

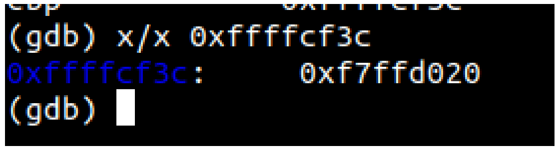


Figure 8: Value in **0xffffcf3c**

The address **0xf7ffd020** is the address in **EBP**.
To verify it, we will see 4 slots of memory from the stack:

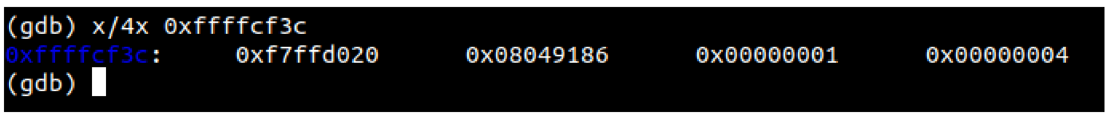
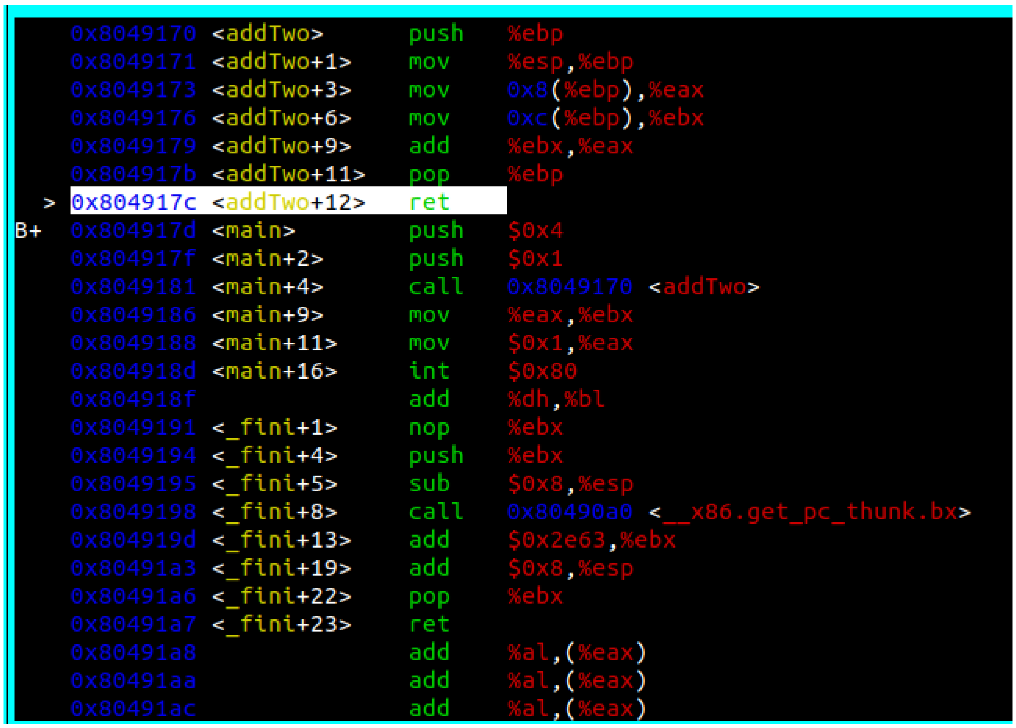


Figure 9: Values in the stack

We can see that the top of the stack contains **0xf7ffd020** which is the address stored in **EBP**. Then we can see the return address **0x08049186** which is the address of the instruction **MOV ebx, eax**, after that we can see the value 1 in hexadecimal and finally we see 4 in hex.

Now, we go on executing the instructions in **addTwo**:

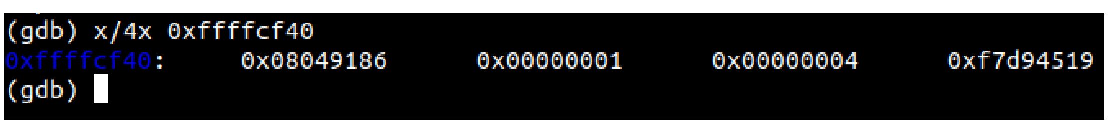


```
    0x8049170 <addTwo>       push   %ebp
    0x8049171 <addTwo+1>     mov    %esp,%ebp
    0x8049173 <addTwo+3>     mov    0x8(%ebp),%eax
    0x8049176 <addTwo+6>     mov    0xc(%ebp),%ebx
    0x8049179 <addTwo+9>     add    %ebx,%eax
    0x804917b <addTwo+11>    pop    %ebp
  > 0x804917c <addTwo+12>    ret
B+  0x804917d <main>         push   $0x4
    0x804917f <main+2>       push   $0x1
    0x8049181 <main+4>       call   0x8049170 <addTwo>
    0x8049186 <main+9>       mov    %eax,%ebx
    0x8049188 <main+11>      mov    $0x1,%eax
    0x804918d <main+16>      int    $0x80
    0x804918f                add    %dh,%bl
    0x8049191 <_fini+1>      nop    %ebx
    0x8049194 <_fini+4>      push   %ebx
    0x8049195 <_fini+5>      sub    $0x8,%esp
    0x8049198 <_fini+8>      call   0x80490a0 <__x86.get_pc_thunk.bx>
    0x804919d <_fini+13>     add    $0x2e63,%ebx
    0x80491a3 <_fini+19>     add    $0x8,%esp
    0x80491a6 <_fini+22>     pop    %ebx
    0x80491a7 <_fini+23>     ret
    0x80491a8                add    %al,(%eax)
    0x80491aa                add    %al,(%eax)
    0x80491ac                add    %al,(%eax)
```

Figure 10: Execution in progress

We can see that we are about to execute the **RET** instruction that is we are about to return back to the caller(main).

Now, we will check what is in the stack:



```
(gdb) x/4x 0xffffcf40
0xffffcf40:     0x08049186      0x00000001      0x00000004      0xf7d94519
(gdb)
```

Figure 11: **EBP** removed from stack

We can see that **EBP** has been removed and **ESP** has been decremented and now the return address is at the top of the stack. Now after executing the **RET** instruction, we will return to the instruction right after the **CALL** instruction in main.

We can also see that the addition was performed and the result is stored in **EAX**:

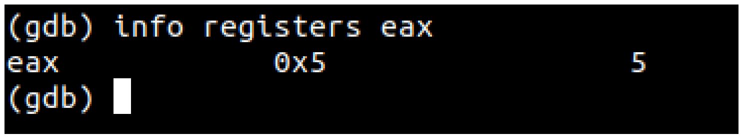


Figure 12: **EAX** stores result