ASM code reading#01

Filovirid
filovirid@protonmail.com

February 1, 2020

1 Question

You can find the question related to this solution here. Read the assembly code and answer the following questions:

1. What does this code do?
2. What are the values in lines 1 and 8?
3. Which functions in C can do the same job as this assembly code?
4. If this is a function, then what are the inputs?
5. Draw the stack diagram related to the function.

2 Objectives

The learning objective of this assembly code is to learn how to read assembly code and also how values and parameters stored in stack. Also to understand the different types of calling conventions and stack clean-up process.

3 Assembly function calls

Before diving into the answer, first, we need to see how calling functions works in assembly, and how function parameters are pushed into the stack.

In this solution, we only explains about x86 (32-bits) codes and operating systems since x64 (64-bits) systems, use a slightly different approach for calling functions.

Let's assume we have the following C source code:

```
//*************Code*************
#include <stdio.h>

int myfunc(int, int);

int main(int argc, char ** argv){
    int a,b,c;
    a = 12;
    b = 15;
    c = myfunc(a,b);
    printf("The result is: %d",c);
    return 0;
}

int myfunc(int a, int b){
    return a + b;
}
//*************End*************
```

The function *myfunc*, accepts two integers as input parameters. We call the function from our main function, with two parameters, a and b:

```
// a = 12 and b = 15
c = myfunc(a, b)
// c = myfunc(12,15)
```

So the first parameter (a) is 12 and the second parameter (b) is 15. Now lets look at the assembly code generated by GCC compiler.

```
;function main
....
01: push 0Fh
02: push 0Ch
03: call myfunc
04: add esp, 8
05: ....
```

```
06:  ....
```

And looking at the assembly code of the *myfunc* we have:

```
;function  myfunc
01: push      ebp
02: mov       ebp, esp
05: mov       edx, [ebp+8]
06: mov       eax, [ebp+0CH]
07: add       eax, edx
08: pop       ebp
09: retn
```

I removed some unnecessary parts to make the code both shorter and easier to understand. So, as you can see, in the main function, before calling *myfunc*, we have to push the parameters into the stack **in reverse order** and then call the function. In other words, while in C we call the function like myfunc(12,15), in assembly we first push 15 into the stack, then push 12 into the stack and finally call the function. This is actually the way that parameters are passed to functions in 32-bit (x86) operating systems. Here, the function which calls another function is the *caller*, and the function which is called is the *callee*. In our example, *main* function is caller and *myfunc* function is the callee.

There are three major calling conventions which are: 1) CDECL, 2) STD-CALL and 3) FASTCALL. The best way to understand the difference between them is to read this post from Wikipedia.

For this tutorial, what important for us is to understand only one major difference between *cdecl* and *stdcall* about stack operation. In *cdecl*, the caller:

1. Push parameters in reverse order into stack
2. Call the function
3. Clean up the stack and remove the pushed parameters

While in *stdcall*:

1. Push parameters in reverse order into stack.
2. Call the function (here the callee is responsible to clean up the stack and remove the pushed parameters)

Referring to our above listed assembly code, line 04 of the *main* function (add esp, 8) is responsible to clean the stack. Therefore, we can say this is a *cdecl* calling convention.

4 Answer

Now that we know a little bit more about how calling functions in assembly works, let's see the code related to the question.

```
01: 8B 7D 08     mov edi, [ebp+8]
02: 8B D7        mov edx, edi
03: 33 C0        xor eax, eax
04: 83 C9 FF     or ecx, 0FFFFFFFFh
05: F2 AE        repne scasb
06: 83 C1 02     add ecx, 2
07: F7 D9        neg ecx
08: 8A 45 0C     mov al, [ebp+0Ch]
09: 8B FA        mov edi, edx
10: F3 AA        rep stosb
11: 8B C2        mov eax, edx
```

As we can see, it is just an 11 line assembly code. While we have no clue what this code does, the best guess is to consider the code as part of the function (as mentioned implicitly in questions). So if we consider this code as a function, we can say two parameters were passed to this function and they are stored in *epp + 8* and *ebp + 0CH*. How do I know that? let's draw the stack and decorate our assembly code to check our hypothesis:

First wrap the given code into a function and create a main function to call the given code.

```
;The following assembly code is used to call
;the function in the question (we name it function f)
;compile it with NASM, GCC and then run it.
;nasm -f elf32 -o test.o test.nasm
;gcc  -o test test.o -m32
;./test
section  .data
mychar db 'Z'
mystr  db    "This is a test string", 0
```

```
szformat db '%s',10,0

section   .text
global main
extern printf
extern exit

main:
01: xor eax,eax
02: mov al, byte [mychar]
03: push eax
04: push dword mystr
05: call f
06: add esp, 8
07: push mystr
08: push szformat
09: call printf
10: add esp, 8
11: mov  eax,0
12: push eax
13: call exit

f:
14: push ebp
15: mov ebp, esp
;;;;;;;;;;;;;;;;;;;;;;;;;
16: mov edi, [ebp+8]
17: mov edx, edi
18: xor eax, eax
19: or ecx, 0FFFFFFFFh
20: repne scasb
21: add ecx, 2
22: neg ecx
23: mov al, [ebp+0Ch]
24: mov edi, edx
25: rep stosb
26: mov eax, edx
;;;;;;;;;;;;;;;;;;;;;;;;
```

```
27: mov esp, ebp
28: pop ebp
29: ret
;;;;;;;;;;;;;;END;;;;;;;;;;;;
```

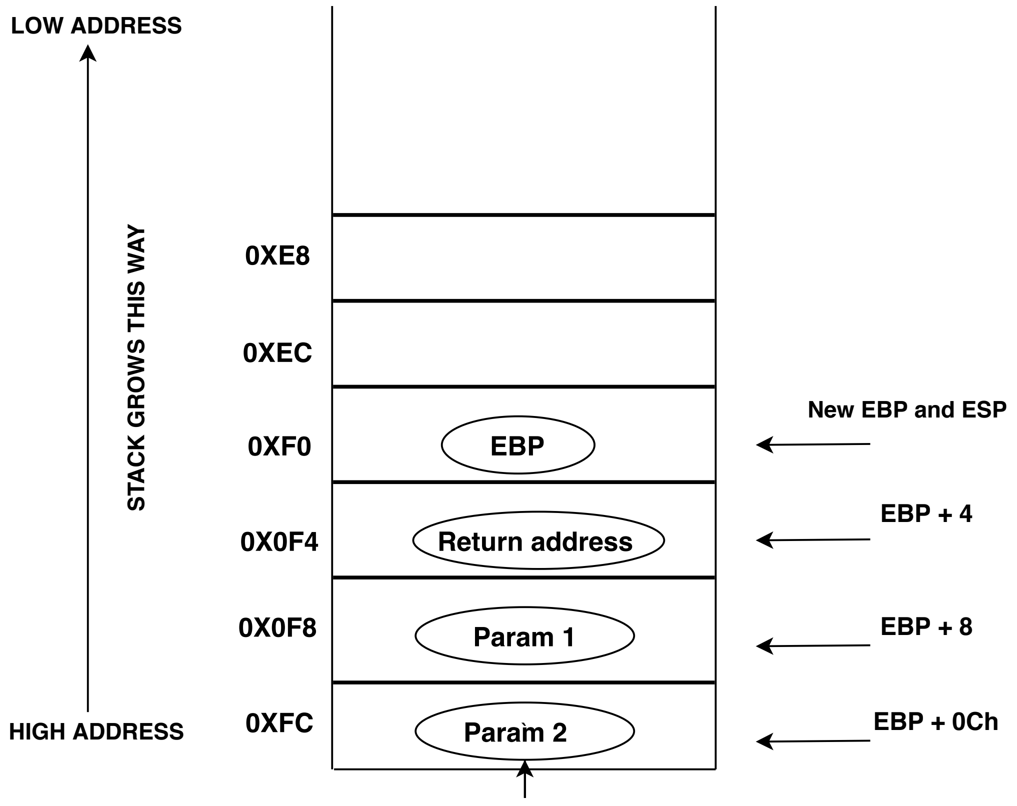

Figure 1: Stack diagram after calling a function with two parameters.

Since we are using *cdecl* calling convention, we push the parameters in the reverse order. Therefore, the first value we push into the stack (line 03) is the last argument of the function which is the value of 'mychar'. The next value we push into the stack (line 04) is actually the first parameter of the function (function has two arguments) and is the address of 'mystr'. Looking at 1, 'param2' in the Figure is equal to line 03 and 'param1' in the Figure is equal to line 04. Then we have 'call f' instruction which push the return address into the stack (address of line 06) and jump to line 14 (set EIP to address of line 14). In line 14, we push the old EBP into stack (address 0xF0 of stack) and we set the new EBP value to the current ESP value. Therefore, both EBP and ESP now points to 0xF0 in Figure 1.

Line 16-20, work like a loop and compare each character of 'mystr' with null value to find the length of the string. If you don't understand how it works, you should read about scasb and repne. This means that now we have the length of the string in ECX. Line 23, copy the second parameter of the function from stack (address 0xFC in Figure 1) to AL. Line 24 and 25 copy the value in AL to each byte of the memory that belongs to 'mystr' and replace the whole string with 'param2'. So this is basically 'memset' function in C.

In total, the whole given code in the question does two things:

1. Calculate the length of the given string.
2. Replace the characters of the string with the given character.

The prototype of the function 'f' is as below:

```
char * f(char * str, char mask);
```

And is equivalent to the following C code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
char * f(char * str, char mask){
        memset((char *)str, (int) mask,strlen(str));
}

int main(){
        char teststr[] = "This is a test string";
        char * str = (char *) malloc(strlen(teststr)+1);
        memcpy(str,teststr,strlen(teststr));
        char mask = 'Z';
        f(str,mask);
        printf("%s\n",str);
        return 0;
}
```