Practical Reverse Engineering Exercises - Write Ups

Chapter 1 – Exercise 1 (6th of July 2014)

TASK

“This function uses a combination SCAS and STOS to do its work. First, explain what is the type of the [EBP+8] and [EBP+C] in line 1 and 8, respectively. Next, explain what this snippet does.

```
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”
```

Excerpt from: *“Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation”,*

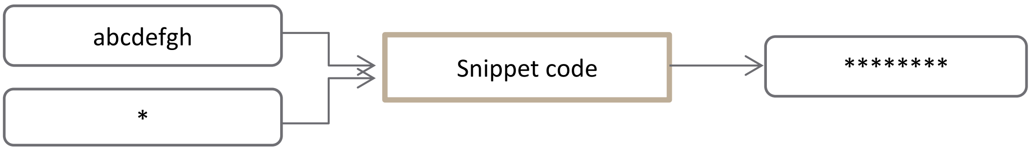
Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, ISBN: 978-1-118-78731-1

MY SHORT ANSWER

EBP+8 is of type pointer to a char (first element of null terminated string)

EBP+C is of type char (1 byte)

The snippet is a body of a function responsible for replacing every character from a given string with another predefined character.

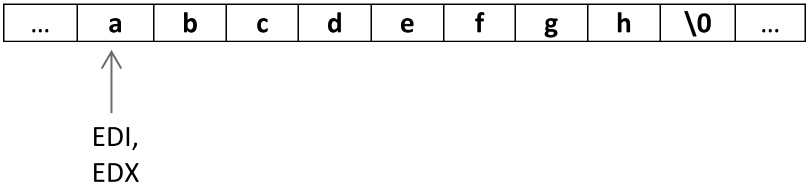


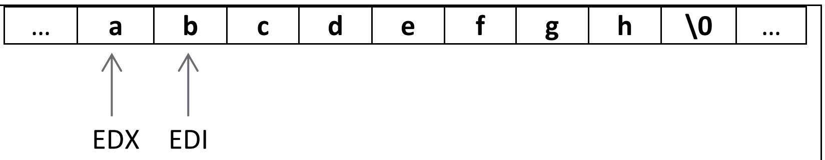
MY INTERPRETATION

Using some initial previous knowledge about how the function calls in assembly work I assumed that the snippet is most probably part of a function body. Those suspicions were fed by the use of EBP register, particularly [EBP+8] and [EBP+0Ch]. Usually those are pointers to the function arguments. The following is a standard representation of the stack layout after a function call.

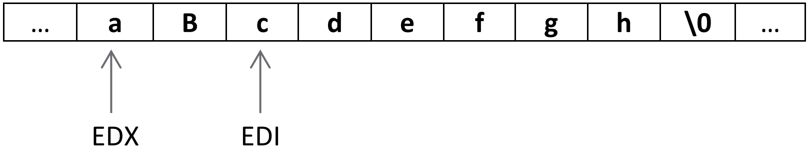
	….	
+Ch	Second function argument	
+8	First function argument	
+4	Old EIP (return address)	
0	Old EBP	← EBP
-4	First local variable	
-8	Second local variable	
	…	

Detailed Intel x86 function calls explanation could be found on http://unixwiz.net/techtips/win32-callconv-asm.html

Line of code	**Explanation**
01: 8B 7D 08 mov edi, [ebp+8]	Load the first function argument to EDI (pointer to our string ***abcdefgh***)
02: 8B D7 mov edx, edi	Store the current EDI value (pointer to our string ***abcdefgh***) into EDX for purpose clarified later
03: 33 C0 xor eax, eax	Set EAX =0
04: 83 C9 FF or ecx, 0FFFFFFFFh	Set ECX to maximum representable value (-1 when viewed as signed value).
05: F2 AE repne scasb	Traverse through every string character (the one pointed by EDI) till Null terminator is reached. That comes as a combination of SCASB instruction which compares EDI and AL values. REPNE repeats SCASB until EDI and AL(0) match or until ECX becomes 0. This operation increments EDI and decrements ECX automatically. **Before the first iteration** **ECX**=0FFFFFFFFh (**-1**)  **After the 1st iteration** **ECX**=0FFFFFFFEh (**-2**)

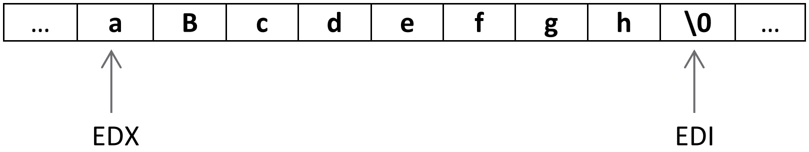
<table>
<tr><td></td><td>


<b>After the 2nd iteration</b>
<b>ECX</b>=0FFFFFFFDh (<b>-3</b>)

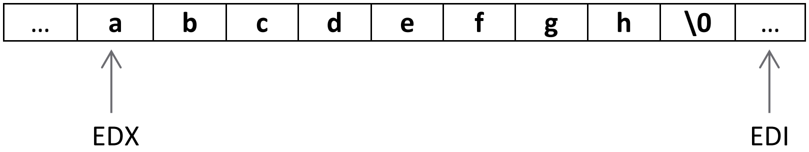


...

<b>After the 8th iteration</b>
<b>ECX</b>=0FFFFFFF7h (<b>-9</b>)



<b>After the 9th iteration</b>
<b>ECX</b>=0FFFFFFF6h (<b>-10</b>)



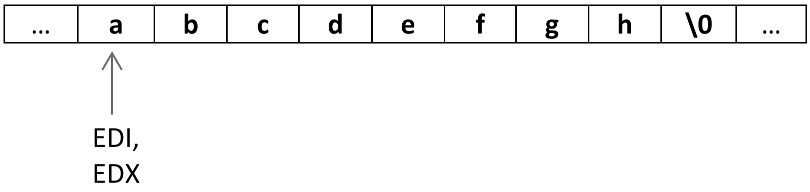
</td></tr>
<tr><td>06: 83 C1 02 add ecx, 2</td><td>
Those 2 lines could be explained as follows.
Before executing the instruction at line 06 the initial string has been traversed including the Null terminator.
Assuming that the string is N characters long (In the case of <b><i>abcdefgh</i></b> N=8) then the ECX has been decreased N +1 times (because of the Null terminator) and taking in account the fact that its initial value was -1 then at that point
ECX = -N -2

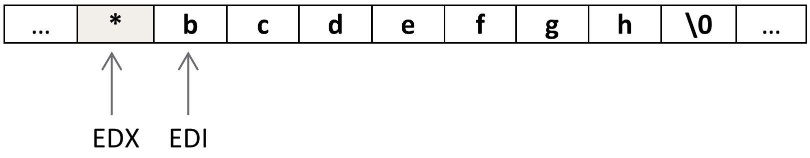
<b>So what exactly is going on in line 06 and 07?</b>

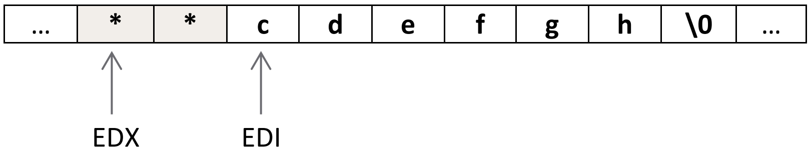
<b>add ecx, 2</b> ;increase the ECX value with 2 i.e. ECX=-N
<b>neg ecx</b> ; ECX=-(-N)=N

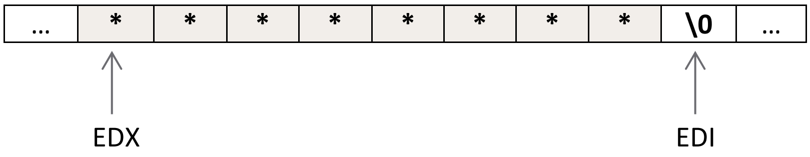
That means that <b>after executing line 06 and 07 ECX will contain the length of the initial string</b> (in our case 8).
</td></tr>
<tr><td>07: F7 D9 neg ecx</td><td></td></tr>
<tr><td>08: 8A 45 0C mov al, [ebp+0Ch]</td><td>Load the second function argument into AL (Our character *). Since AL register is used we can safely assume that the size of the argument is 1 byte.</td></tr>
</table>

<table>
<tr><td>09: 8B FA</td><td>mov edi, edx</td><td>Load what is stored in EDX to EDI. Looking back at line02 we see that EDX stores pointer to the first function argument (our initial string <b><i>abcdefgh</i></b>)</td></tr>
<tr><td>10: F3 AA</td><td>rep stosb</td><td>STOSB reads AL value (*) and stores it at the address pointed by EDI (first character of our <b><i>abcdefgh</i></b> string). REP will execute STOSB until ECX value becomes 0. After each execution EDI is incremented and ECX is decremented automatically. Before executing this instruction ECX=N which means that the operation will be repeated N times (in our case 8 times)

<b>Before the first iteration</b>
<b>ECX=8</b>


<b>After the 1st iteration</b>
<b>ECX=7</b>


<b>After the 2nd iteration</b>
<b>ECX=6</b>


…
<b>After the 8th iteration</b>
<b>ECX=0</b>

</td></tr>
<tr><td>11: 8B C2</td><td>mov eax, edx”</td><td>EDX points to the first character of the modified string. Usually the result of a function is stored in EAX register. That line of code supports the initial theory that the snipped is part of a function body.</td></tr>
</table>

PROOF OF CONCEPT

I decided to write an assembly program and run it through IDA Debugger to check if my interpretation is holding up which it did.

You could find the source code of this and more of my solutions at the dedicated github project - https://github.com/malchugan/PRE-Exercises

Note: this is a Linux assembly file using AT&T syntax. To assemble and link the code bellow execute the following form the command line:

$as -gstabs -o ex1_att.o ex1_att.s

$ld -o ex1_att ex1_att.o

As a result you should have ex1_att executable which you could examine with GDB, IDA or any other debugger of your choice

ex1_att.s listing

```
.data
        myString:
                .asciz "abcdefgh"

.text
        .globl _start
        .type TestFunc, @function

        TestFunc:
                # function prologue
                push %ebp               # preserve the old EBP value
                movl %esp, %ebp

                # function body - same as example 1 code in AT&T syntax
                movl 8(%ebp), %edi      # load the first function parameter in EDI - pointer to myStirng
                movl %edi, %edx         # store the initial EDI value in EDX
                xor %eax, %eax          # EAX = 0
                or  $0xffffffff, %ecx   # set ECX to the maximum representable value
                repne scasb             # compare EDI content byte by byte with AL (NULL) or ECX becomes 0 :)
                add $2, %ecx            # at the end of repne scasb ECX has value (-strlen -2). With this sum ECX = -
strlen
                neg %ecx                # ECX = strlen
                movb 12(%ebp), %al      # load second function argument
                movl %edx, %edi         # load the initial EDI value (firts argument) back to EDI
                rep stosb               # store strlen number of bytes in EDI al with the al value (character)
                movl %edx, %eax         # store EDX value in EAX

                # function epilogue
                movl %ebp, %esp                 # restore the old ESP value
                pop %ebp                # restore the old EBP value
                ret

        _start:
                # push the function arguments into the stack
                push $0x2A              # ASCII '*'
```

```
    push $myString              # pointer to myString

    # call the funciton
    call TestFunc

    # exit gracefully
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```