

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, 0FFFFFFFh
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](https://www.amazon.com/dp/9781118787311)

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, par-

Practical Reverse Engineering Exercises - Write Ups

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

ticularly [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>

SNIPPET WALKTHROUGH

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, 0FFFFFFFh	Set ECX to maximum representable value (-1 when viewed as signed value).											
05: F2 AE repne scasb	<p>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.</p> <p>This operation increments EDI and decrements ECX automatically.</p> <p>Before the first iteration ECX=0FFFFFFFh (-1)</p> <table><tr><td>...</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td><td>h</td><td>\0</td><td>...</td></tr></table> <p>↑ EDI, EDX</p>	...	a	b	c	d	e	f	g	h	\0	...
...	a	b	c	d	e	f	g	h	\0	...		

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

Page 3 of 6

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

Page 4 of 6

Practical Reverse Engineering Exercises - Write Ups

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

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 from 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 myString
        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 (first 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 '*'
```

Practical Reverse Engineering Exercises - Write Ups

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

```
push $myString      # pointer to myString
```

```
# call the function  
call TestFunc
```

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