The Vulnerable Space

Wednesday, 29 July 2015

(N)ASM101 - 0x03 - Improving strlen()

Previously we've looked at how Win32 API functions are called and how to create our own function which computes the length of a string and stores the result in a register. Some people have pointed out to me that my construction of strlen() is very long-winded. The reason I did this is that the implementation I presented uses simpler ASM instructions and I didn't want to overwhelm readers who are new to this field. On the other hand, now is a good time to introduce these improvements.

x86 Instructions

When it comes to repetitious data movement and comparison, the Intel instruction set provides us with native instructions which faciliate these operations. These conventient instructions will be presented later in this section but first let's cover a few new ASM instructions which will be useful in constructing the improved version of strlen().

NOT

The NOT instruction performs a bitwise NOT of a register or a memory location and stores it in that same location. A logical NOT essentially inverts input bits: NOT 1 = 0 and NOT 0 = 1. Inverting the bits of an integer value is called computing the Ones' Complement of an integer. The following are 2 examples of such:

```
+-----------------------------------------------------------------+----------+----------+
|  +/-  |       |       |       |      |      |      |      | 1s' Comp | Unsigned |
|-------+-------+-------+-------+------+------+------+------+----------+----------|
|   0   |   0   |   0   |   0   |  0   |  0   |  0   |  0   |    0     |    0     |
|-------+-------+-------+-------+------+------+------+------+----------+----------|
|   0   |   1   |   1   |   1   |  1   |  1   |  0   |  1   |   125    |   125    |
+-------+-------+-------+-------+------+------+------+------+----------+----------|
|   1   |   0   |   0   |   0   |  0   |  0   |  1   |  0   |  -125    |   130    |
|-------+-------+-------+-------+------+------+------+------+----------+----------|
|   1   |   1   |   1   |   1   |  1   |  1   |  1   |  1   |   -0     |   255    |
+-----------------------------------------------------------------+----------+----------+
```

This operation is synonymous to XORing with 0FFFFFFFFh, since 0 XOR 1 = 1 and 1 XOR 1 = 0, and to subtracting from 0FFFFFFFFh, where 0FFFFFFFFh is the 1s' complement representation of -0.

An evident issue with this representation is that there are 2 representations of 0. Also, the addition of 2 numbers in their 1s' complement representation is not always equal to the 1s' complement representation of the addition of the 2 numbers in decimal form. This irregularity happens when the result overflows and has to be accounted for with a carry:

```
  0001 1111     31 +
- 1111 1000     -7
-----------    ----
1 0001 0111     23   <-- Overflow
+ 0000 0001      1   <-- End-around carry
-----------    ----
  0001 1000     24   <-- Result: 31 + (-7)
```

These issues have been circumvented by using the Two's complement representation system which will be discussed in the next subsection.

NEG

NEG performs the Two's Complement of the operand's value and overwrites it with the newly computed value. The Two's Complement of a binary number is obtained in the following manner:

1. Scan from right to left until the 1st "1" is found
2. Perform Ones' Complement on the bits to the left of the "1"

OR:

1. Perform Ones' Complement
2. Add 1

As an example:

```
+-----------------------------------------------------------------+----------+----------+
```

Contact

Follow @GradiusX

Blog Archive

▼ 2015 (5)
► September (1)
▼ July (2)
(N)ASM101 - 0x03 - Improving strlen()
(N)ASM101 - 0x02 - Constructing strlen()
► June (2)

-128	64	32	16	8	4	2	1	2s' Comp	Unsigned
0	0	1	0	1	1	0	0	44	44
1	1	0	1	0	1	0	0	-44	212

The advantages of this system is that there's only 1 representation of 0 and the addition, subtraction or multiplication operations are identical to those for unsigned binary numbers. An overflow is simply discarded. Let's go through the previous example using the 2s' Complement system:

```
   0001 1111     31 +
 - 1111 1001     -7
 -----------    ----
 1 0001 1000     24   <-- Discard Overflow
 -----------    ----
   0001 1000     24   <-- Result: 31 + (-7)
```

REPZ(REPE), REPNZ(REPNE) & REP

The REP instruction and its variations are used to repeat operations up to ECX times. Each time the secondary instruction is performed, ECX is decremented. Repeat while Zero (REPZ) and Repeat while Equal (REPE) are interchangeable and keep executing the input instruction while ECX != 0 and ZF = 1. Repeat while Not Zero (REPNZ) and Repeat while Not Equal (REPNE) execute the input instruction while ECX != 0 and ZF != 1.

We'll look at examples at how these can be combined with SCAS to deal with repetitious comparisons in the next subsection.

SCAS

SCAS comes in 3 flavours: SCASB, SCASW, SCASD, that operate at 1-,2-, or 4-byte granularity respectively. SCAS implicitly compares AL/AX/EAX with data starting at the memory address EDI; EDI is automatically incremented/decremented depending on the Direction Flag (DF).

Consider the following example:

```
mov edi, input       ; assume input is a user defined string
mov ecx, 05h         ; move 0x5 to ecx
mov al, 041h         ; move 0x41 to al (0x41 = 'A')
repne scasb          ; repeatedly compares until ECX = 0 or 'A' is reached
```

Let "input" point to the start of an arbitrary string. The program will compare this string, a byte at a time, with 'A', each time decremeting ECX. The program terminates either when ECX reaches 0 or when an 'A' is found since the latter will set the Zero Flag, whichever comes first. At this point you've probably figured out that the combination of REP and SCAS will be at the center of the solution to improving strlen(), and hence of this tutorial.

The New and Improved strlen()

We now put the newly-learnt techniques to good use:

```
;nasm -fwin32 strlen_improved.asm
;GoLink /entry _main strlen_improved.obj
;Run under a debugger

global _main

section .data
    input db "What is the length of this string?",0    ; string to compute length on

section .text
_main:
    xor    ecx, ecx       ; ecx = 0x00000000
    not    ecx            ; initialize ecx to largest value possible (4,294,967,295 in 32-bit)
    xor    al, al         ; al = 0x00
    mov    edi, input     ; edi points to start of string
    mov    ebx, edi       ; store original pointer
    repne  scasb          ; repeatedly compare bytes
    sub    edi, ebx       ; subtract to get length + 1
    dec    edi            ; decrement edi
done:
    int    3              ; debugger interrupt
```

not ecx

Performs a NOT on 0x00000000, returning 0xFFFFFFFF which is 4,294,967,295 in decimal form, the highest possible 32-bit unsigned integer. This is to avoid REPNE from quitting due to ECX hitting zero before reaching the end of the string.

xor al, al

Zeros out AL which will be used by SCASB to compare bytes from the input string with. Remember that to find the length of the string we need to iterate over each byte until the NULL terminator is encountered

mov ebx, edi

Stores the address of the start of the string. This will be used to calculate the final result.

repne scasb

Roughly translates to : while (ZF != 0 OR ECX != 0) { compare byte with 0x00; increment EDI; decrement ECX}. The Zero Flag is set when AL and the current character that is being compared to are equal, i.e. when 0x00 is encountered.

sub edi, ebx

Subtracts the position of the NULL from the start of the string.

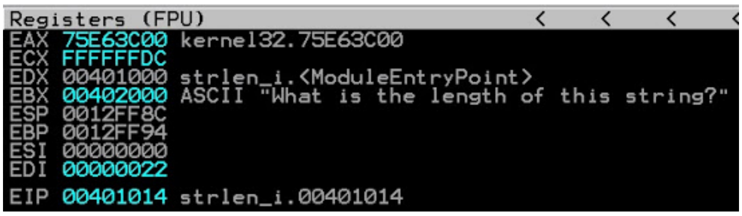
dec edi

Decrements the result to account for NULL.

Generate the executable using NASM and GoLink:

```
Command Prompt

C:\>nasm -fwin32 strlen_improved.asm

C:\>GoLink /entry _main strlen_improved.obj

GoLink.Exe Version 1.0.1.0 - Copyright Jeremy Gordon 2002-2014 - JG@JGnet.co.uk
Output file: strlen_improved.exe
Format: Win32   Size: 1,536 bytes

C:\>
```

Attach a debugger and look at the result saved in EDX.



Further improvements ?

Of course !! Saving the original pointer to the string, subtracting the new value from it, and decrementing it again is not the most efficient way of obtaining the final result.

At the start of the program we've initialized ECX to all 1s, i.e. the maximum unsigned integer, but this can also be interpreted as -1 from a signed-integer point of view. Also, each REPNE iteration decremented this value by 1. At the end we get ECX = - strlen - 2. Why -2? One from the counted string terminator and another from ECX being initialized to -1.

An interesting property of NOT is that given any negative number X, NOT X = |X| - 1. Combining these 2 facts we can calculate the length of the string directly from ECX in the following manner:

```
not  ecx  ; ecx = - strlen - 2, i.e. not ecx = |- strlen - 2| - 1  = strlen + 2 - 1  =  strlen + 1
dec  ecx  ; ecx =   strlen + 1, i.e. dec ecx = strlen !!
```

Adding this new improvement we get :

```
;nasm -fwin32 strlen_improved_2.asm
;GoLink /entry _main strlen_improved_2.obj
;Run under a debugger

global _main

section .data
    input db "What is the length of this string?",0    ; string to compute length on

section .text
_main:
    xor     ecx, ecx       ; ecx = 0x00000000
    not     ecx            ; initialize ecx to -1 (signed int)
    xor     al, al         ; al = 0x00
    mov     edi, input     ; edi points to start of string
    repne   scasb          ; repeatedly compare bytes
    not     ecx            ; ecx = strlen + 1
    dec     ecx            ; account for 0x00
done:
    int     3              ; debugger interrupt
```

Conclusion

Congratulations to those of you who have made it through all the tutorials thus far. As a teaser, in the next post we'll combine today's ASM program with the one presented in the first to construct a finalized, presentable version of strlen() .. バイバイ

Posted by Francesco Mifsud at 12:23 G+1 Recommend this on Google

No comments:

Post a Comment

Enter your comment...

Comment as: Google Account

Publish Preview