

The Vulnerable Space

Tuesday, 14 July 2015

(N)ASM101 - 0x02 - Constructing strlen()

In the previous post we've looked at some of the most commonly used registers, the stack, some x86 instructions, and how to call a higher level Win32 API function. We've combined these to create an ASM program which displays a message box with a custom message and title.

In this post we'll look at some more interesting x86 instructions and construct the strlen() C++ function from scratch.

strlen()

The function expects a pointer to a null-terminated string and returns the length, i.e. the number of characters that make up the string without including the terminating null character. As a trivial example let's consider the string "HELLO" shown below:



As portrayed in the image, the string "HELLO" constitutes of 5 characters and the NULL character, hence passing a pointer to this string to strlen() returns 5.

x86 Instructions

To keep the theory at a minimum, only instructions required to achieve our goal, and some of their variations, will be covered in this section.

MOVE

The MOV instruction is probably the most common ASM instruction. As the name suggests it moves data from one place to another. To be more precise it copies data across rather than move it, since the source value remains intact. The operands can be a register (e.g. EAX), a memory location (e.g. [EAX]), the memory location pointed at by EAX or an immediate value (e.g. 0FFh). The following are some examples of such:

```
1  mov  eax, F003h      ; immediate -> reg ; EAX = 0xF003 ?
2  mov  [eax], F003h    ; immediate -> mem ; [EAX] = 0xF003
3  mov  ebx, eax        ; reg -> reg      ; EBX = EAX
4  mov  eax, [ebx]      ; mem -> reg      ; EAX = [EBX]
5  mov  [ebx], eax      ; reg -> mem      ; [EBX] = EAX
6  mov  eax, [eax+ebx*4] ; reg -> reg      ; EAX = [EAX+EBX*4]
```

Notice that data moves from right to left. This is the Intel syntax, and we'll be sticking exclusively to it. For those of you who are interested, the other major syntax is the AT&T syntax and is predominantly used in *nix environments.

ADD/SUB & INC/DEC

Arithmetic operations can be performed on registers and memory locations. We can add two registers with ADD, subtract an immediate value from a memory location with SUB, increment values with INC and decrement values with DEC.

```
1  add  esp, 14h        ; esp = esp + 0x14 ?
2  add  ecx, eax        ; ecx = ecx + eax
3  sub  ecx, eax        ; ecx = ecx - eax
4  sub  esp, 0Ch        ; esp = esp - 0xC
5  inc  eax             ; eax = eax + 1
6  dec  eax             ; eax = eax - 1
7  inc  dword [eax]     ; [eax] = [eax] + 1
8  add  word [eax], 0FFFFh ; [eax] = [eax] + 0xFFFF
9  add  dword [eax], 0FFFFh ; [eax] = [eax] + 0xFFFF
```

Notice that when the destination is a memory location, the size of the memory that we want to deal with has to be explicitly specified. Let's see why this is important.

Consider the last 2 of the examples above which at first glance look deceptively equivalent. Say that EAX points to an arbitrary location in memory. Also say that if we read a dword in length starting from this location, we retrieve 0x41424344. This means that if we read a word in length starting from the same location, we get 0x4344. So:

```
add dword [eax], 0FFFFh => dword [eax] = 0x41424344 + 0xFFFF
                        => dword [eax] = 0x41434343 & word [eax] = 0x4343 & CF=0
```

```
add word [eax], 0FFFFh  => word [eax] = 0x4344 + 0xFFFF
                        => dword [eax] = 0x41424343 & word [eax] = 0x4343 & CF=1
```

Notice the differences. For the word-case, the most significant part is untouched since we explicitly told it to consider a word in length and the Carry Flag has been set to 1 as the result is larger than a word.

OR/XOR

These instructions perform bitwise (eXclusive)OR of the operands and place the result in the first operand specified. The following are some self-explanatory examples:

Twitter

@GradiusX

Blog Archive

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

```

1 | or ebx, ebx          ; ebx = ebx | ebx
2 | xor eax, [ecx]       ; eax = eax ^ [ecx]
3 | or [edx], eax        ; [edx] = [edx] | eax
4 | xor eax, 0fh         ; eax = eax ^ 0xF
5 | or word [eax], 0FFFFh ; [eax] = [eax] | 0xFFFF

```

?

As in the previous case, when dealing with memory locations, the size has to be explicitly specified. XOR has 2 interesting properties: XORing twice with the same value yields the original value and XORing a value with itself results in zeroes. The latter is used to clear a register whereas the former is sometimes used as a simple shellcode obfuscating technique to evade Antivirus programs.

TEST & CMP

TEST performs a bitwise AND but does not save the result and CMP performs a SUB without saving the result. These operations are used to set the appropriate flags for subsequent operations which may perform different actions depending on these flags.

```

1 | cmp eax, ecx          ; eax - ecx
2 | test [ebx], eax       ; [ebx] & eax
3 | cmp edx, 0FFh         ; edx - 0xFF
4 | test dword [eax], 0ABCDh ; [eax] & 0xABCD

```

?

Once again make sure the length is specified when dealing with memory locations.

JZ(JE), JNZ(JNE) & JMP

Jumps are used to control the flow of an ASM program just like "if..then..else" and "switch" for higher-level programming languages. Jump if Zero (JZ) and Jump if Equal (JE) are interchangeable since the Zero Flag is set to 1 when two equal values are compared, and same for Jump if Not Zero (JNZ) and Jump if Not Equal (JNE). JMP is an unconditional jump and transfers the control flow irrelevant of the flag values.

```

1 | jmp procedure ; jump to "procedure"
2 | je procedure ; jump to "procedure" if ZF=1
3 | jz procedure ; jump to "procedure" if ZF=0

```

?

INT

INT generates a software interrupt and takes a single hexadecimal value. We will not go through the numerous available interrupts but we talk briefly on the following: 0x21 and 0x3. INT 0x21 calls an MS-DOS API call depending on the values in the registers. INT 0x3 (0xCC) is used by debuggers to set a breakpoint.

```

1 | int 3 ; execution breaks here if attached to a debugger

```

?

A complete Intel Syntax Reference guide can be found [here](#).

Constructing strlen()

We've looked at how strlen() works and familiarized ourselves with more than enough ASM instructions to reconstruct the function from scratch. strlen() may be very simple and straightforward but let us decompose it into smaller, digestible operations. This will help us understand the underlying logic and will make it easier to translate to ASM.

strlen() Algorithm:

1. Set CTR = 0
2. Is char at position CTR = 0x00 ?
3. If Yes GOTO 6
4. Increment CTR
5. GOTO 2
6. DONE: Return CTR

Without further ado:

```

1 | ;nasm -fwin32 strlen.asm
2 | ;GoLink /entry _main strlen.obj
3 | ;Run under a debugger
4 |
5 | global _main
6 |
7 | section .data
8 |     input db "What is the length of this string?",0 ; string to compute length on
9 |
10 | section .text
11 | _main:
12 |     xor ecx, ecx          ; clear registers to be used
13 |     xor ebx, ebx          ;
14 |     mov edi, input        ; move pointer to start of input to edi
15 | check_next:
16 |     mov bl, [edi+ecx]     ; move char to bl
17 |     test bl, bl           ; check if char = 0x00 (i.e. end of string)
18 |     jz done              ; if ZF=1 (i.e. bl=0x00), jump to done
19 |     inc ecx              ; increment ecx (counter)
20 |     jmp check_next        ; jmp to check_next
21 | done:
22 |     int 3                ; debugger interrupt

```

?

xor ???, ???

Zeroes out the registers before use. ECX is used both as a counter and a pointer to the next character while the lower part of EBX holds the character we are comparing to.

mov edi, input

Moves the pointer to the start of the string to EDI, i.e. [EDI] = "W".

check_next

A symbolic label used as a reference by jump instructions.

mov bl, [edi+ecx]

Moves the char pointed to by [EDI+ECX] to the BL register.

test bl, bl

Reminder: TEST performs a bitwise AND without storing the result. The only value that returns zero when ANDed to itself, is zero. So, if BL = 0x0 then BL & BL = 0x0, hence ZF=1. If BL <> 0x0, then BL & BL <> 0x0, hence ZF=0.

jz done

If ZF=1, i.e. the previous result was 0x0, i.e. we hit the end of the string, jump to the "done" label.

inc ecx

Increment ECX which holds a counter to the string length.

jmp check_next

Jump to the "check_next" label to repeat the process with an incremented counter and a pointer to the next character.

int 3

A breakpoint for debuggers. This will give us the opportunity to take a look at the ECX register which, at the point it reaches the interrupt, should contain the length of the string in hexadecimal format.

Generate the executable using NASM and GoLink:

Command Prompt

```
C:\>nasm -fwin32 strlen.asm

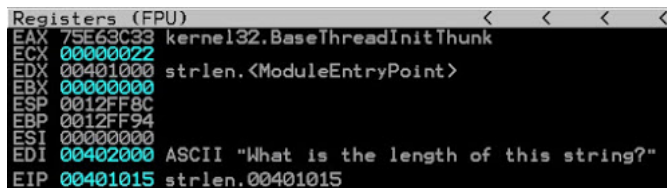
C:\>GoLink /entry _main strlen.obj

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

C:\>
```

As you might have realized by now, we can't just run the executable. We need to attach it to a debugger to be able to view the result in the ECX register. In a future post I might demonstrate how to format a registry value and display it in a message box. It's more involved than it sounds.

The following screenshot shows the status of the registers after the debugger hits the INT instruction:



Register	Value	Comment
EAX	75E63C33	kernel32.BaseThreadInitThunk
ECX	00000022	
EDX	00401000	strlen.<ModuleEntryPoint>
EBX	00000000	
ESP	0012FF8C	
EBP	0012FF94	
ESI	00000000	
EDI	00402000	ASCII "What is the length of this string?"
EIP	00401015	strlen.00401015

ECX contains 0x22 which translates to 34, the length of "What is the length of this string?". Play around with it; modify the string and run it step by step to get a better feel for ASM.

Conclusion

A lot was covered in this post. We've looked at how higher-level functions are constructed using basic assembly instructions. In the next post we won't be creating anything new but rather we'll be improving upon what we've covered here.. さようなら

Posted by Francesco Mifsud at 15:04

 Recommend this on Google

No comments:

Post a Comment

Enter your comment...

Comment as: Google Account ▼

Subscribe to: [Post Comments \(Atom\)](#)