The Vulnerable Space

Saturday, 12 September 2015

(N)ASM101 - 0x04 - Reversing Binaries - Part I (EasyCrack.exe)

In this tutorial we'll temporarily deviate from creating our own ASM programs and try to crack a binary which was created by someone else. Writing assembly is one skill, understanding it is another. The binary we'll be analysing can be found here. The keygen was created by Kwazy Webbit from the Reverse Engineering Team.

x86 Instructions

As with every tutorial we'll first cover the new instructions we'll be encountering during this reverse engineering task.

AND

The AND operator performs a bitwise AND of the operands and places the result in the first operand specified. To refresh your memory, the AND operator only returns TRUE if both operands are TRUE. The following is the truth table for this operator:

```
+-----------+-------------------------------+
|     A     |   0   |   0   |   1   |   1   |
|-------------------------------------------|
|     B     |   0   |   1   |   0   |   1   |
+-----------+-------------------------------+
|   A & B   |   0   |   0   |   0   |   1   |
+-----------+-------------------------------+
```

Some examples:

```
and eax, ebx              ;  EAX  =  EAX  &  EBX
and eax, 0Fh              ;  EAX  =  EAX  &  0xF
and [esi], esp            ; [ESI] = [ESI] &  ESP
and [esi], dword 0ABCh    ; [ESI] = [ESI] & 0x0ABC
```

LEA

LEA stands for Load Effective Address and is one of the data movement instructions. Let's look at a few examples:

```
lea eax, [ebp+0C0h]       ; if EBP = 0x0000ABCD, then EAX = 0x0000AC8D
lea eax, [ebx+ebx]        ; if EBX = 0x00001234, then EAX = 0x00002468
lea ebx, [eax+ecx*4]      ; if EAX = 0x00000010 & ECX = 0x0000000A then EBX = 0x00000038
```

The syntax used for LEA is quite misleading; even though the square brackets are used, the instruction does not reference memory but rather it evaluates the contents of the square brackets directly.

MOVZX & MOVSX

These instructions are special versions of the MOV instruction which are used to extended signed (MOVSX) and unsigned (MOVZX) registers to larger registers.

MOVZX stands for Move With Zero Extension. When data is moved from a register to a larger register, the value is prepended with zeroes. Some examples are:

```
movzx eax, bx       ; if BX = 0x1000, then EAX = 0x00001000
movzx ebx, ax       ; if AX = 0xFFFF, then EBX = 0x0000FFFF
movzx ax, bl        ; if BL = 0xFF, then AX = 0x00FF
movzx bx, al        ; if AL = 0x7F, then BX = 0x007F
```

MOVSX stands for Move with Sign Extension. This time the resultant value is prepended with either zeros or ones depending on the sign of the original value, and hence the sign is preserved. Let's take the previous examples:

```
movsx eax, bx       ; if BX = 0x1000, then EAX = 0x00001000
movsx ebx, ax       ; if AX = 0xFFFF, then EBX = 0xFFFFFFFF
movsx ax, bl        ; if BL = 0xFF, then AX = 0xFFFF
movsx bx, al        ; if AL = 0x7F, then BX = 0x007F
```

ROL & ROR

ROL stands for Rotate Left whereas ROR stands for Rotate Right. These instructions shift the bits of the destination to the left/right a number of times; the data that slides off the end of the register is inserted back from the right/left. The following table should clear things up:

Contact

Follow @GradiusX

Blog Archive

EAX	1100 0011 1010 1110 0001 1111 0000 0101	0xC3AE1F05
ROL EAX, 01h	1000 0111 0101 1100 0011 1110 0000 1011	0x875C3E0B
ROL EAX, 06h	1110 1011 1000 0111 1100 0001 0111 0000	0xEB87C170
ROL EAX, 0Fh	0000 1111 1000 0010 1110 0001 1101 0111	0x0F82E1D7
ROR EAX, 01h	1110 0001 1101 0111 0000 1111 1000 0010	0xE1D70F82
ROR EAX, 06h	0001 0111 0000 1110 1011 1000 0111 1100	0x170EB87C
ROR EAX, 0Fh	0011 1110 0000 1011 1000 0111 0101 1100	0x3E0B875C

It stands to reason that rolling a 32-bit register 33 times is the same as rolling it in the same direction once, hence if say the value of a register has to be rolled X times, where X is a large number, instead roll it X % 32 times.

MUL (IMUL) & DIV (IDIV)

As you've probably guessed MUL performs multiplication while DIV performs Division; IMUL and IDIV are the signed versions.

Although in real life these are basic operations, they might be tricky to handle for the CPU which has a limited number of fixed-sized registers to work with. This is because when multiplying 2 large numbers, the result can be too large to fit into a single register; when dividing, the result can be a decimal number with a decimal part which is too long to fit into a single register. To solve this issue, 2 registers are used, namely EAX, the main register, and EDX.

Let's see some examples on how these 2 registers are used to store the results:

```
mul  ecx          ; EDX:EAX = EAX * ECX (unsigned)
imul edx          ; EDX:EAX = EAX * ECX (signed)

imul ecx, 1Fh     ; ECX = ECX * 0x1F (signed)

div ecx           ; EAX = EDX:EAX / ECX (unsigned)
                  ; EDX = EDX:EAX % ECX (unsigned)

div cl            ; AL  = AX / CL (unsigned)
                  ; AH  = AX % CL (unsigned)

idiv ecx          ; EAX = EDX:EAX / ECX (signed)
                  ; EDX = EDX:EAX % ECX (signed)
```

Notice how both of these functions can make use of EAX, the accumulator register, without explicit declaration. As mentioned earlier, the EDX register is used as a secondary storage for overflows in multiplications and remainders in divisions. Another interesting thing to notice is that MUL always operates and stores the result in EAX/AX/AL and hence the following instruction is invalid:

```
mul  ecx, 1Fh    ; ECX = ECX * 0x1F  <- INVALID
```

SBB

SBB is Subtract with Borrow/Carry. It differs slightly from the traditional subtract, i.e. SUB, in that if the Carry Flag is set, it subtracts 1 extra, otherwise it's exactly the same. For example:

```
sbb  ax, 10h (CF = 1)    ; if EAX was 0x0050, then EAX = 0x003F
sbb  ax, 10h (CF = 0)    ; if EAX was 0x0050, then EAX = 0x0040
```

JA(JG), JAE(JGE), JB(JL) & JBE(JLE)

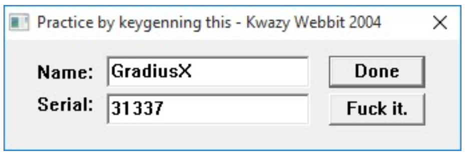
These operators are different conditional jump instructions which operate on either signed or unsigned values. The following is a summary table which contains a self-explanatory description and the jump conditions for each:

Mnemonic	Meaning	Jump Condition
JA	Jump if Above	CF=0 & ZF=0
JAE	Jump if Above or Equal	CF=0
JB	Jump if Below	CF=1
JBE	Jump if Below or Equal	CF=1 | ZF=1
JG	Jump if Greater (signed)	ZF=0 & SF=OF

```
|------------|----------------------------------------+--------------------|
|    JGE     |   Jump if Greater or Equal (signed)    |       SF=OF        |
|------------|----------------------------------------+--------------------|
|     JL     |         Jump if Less (signed)          |       SF!=OF       |
|------------|----------------------------------------+--------------------|
|    JLE     |     Jump if Less or Equal (signed)     |   ZF=1 | SF!=OF    |
+------------+----------------------------------------+--------------------+
```

The Binary

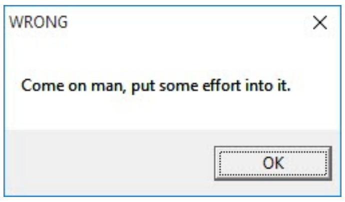
Finally we've made it to the interesting bit. The binary presents us with a tiny window and 2 text fields named **Name** and **Serial**, the latter of which can only be numerical. Let's try our luck ...


Practice by keygenning this - Kwazy Webbit 2004

Name: GradiusX

Serial: 31337

... anddddd ...

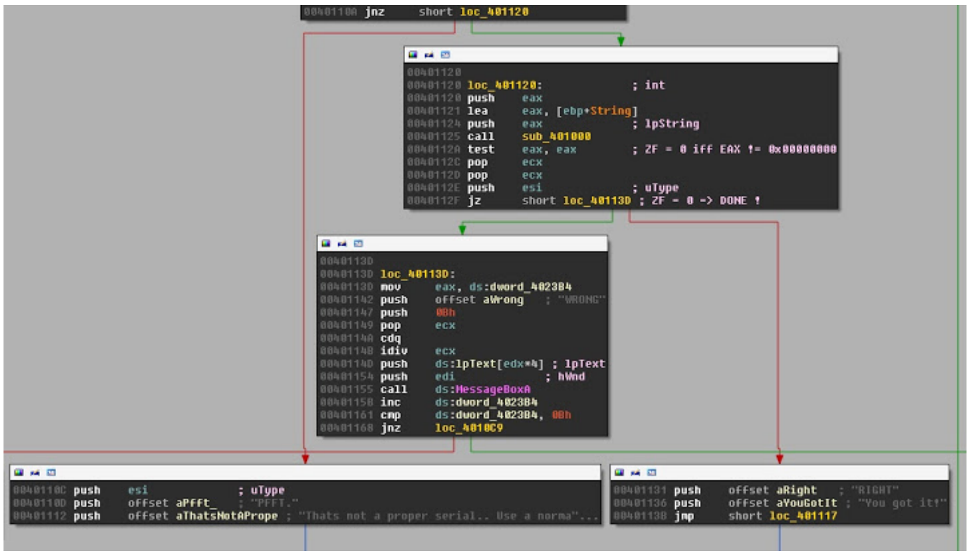

WRONG

Come on man, put some effort into it.

... nope nothing. At this point we don't know if it is expecting a specific **Name** and **Serial**, or maybe a specific **Name** and a derived **Serial** or even any **Name** and a **Serial** number which is derived from it. So let's look at it under IDA.

Identifying Important Areas

Looking at the strings present in the binary, we notice 2 very interesting ones, "RIGHT" and "You got it!". We know that that's where we need to end up so lets start from there. My personal strategy when tackling binaries in which I known what the point I want to reach is, is to start from there and work my way up as much as I can. Let's apply this.

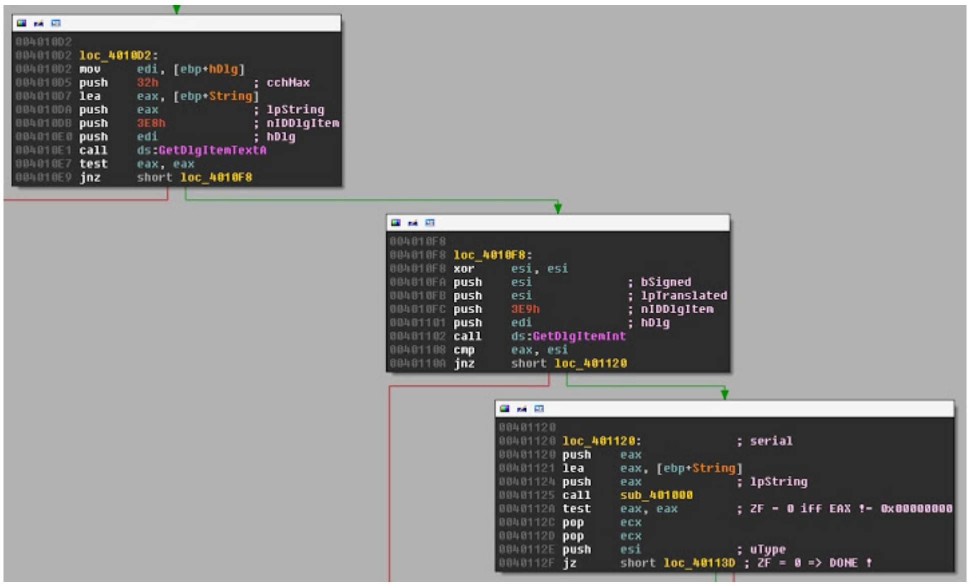


```
0040110A jnz     short loc_401120

00401120
00401120 loc_401120:                     ; int
00401120 push    eax
00401121 lea     eax, [ebp+String]
00401124 push    eax                     ; lpString
00401125 call    sub_401000
0040112A test    eax, eax                ; ZF = 0 iff EAX != 0x00000000
0040112C pop     ecx
0040112D pop     ecx
0040112E push    esi                     ; uType
0040112F jz      short loc_40113D ; ZF = 0 -> DONE !

0040113D
0040113D loc_40113D:
0040113D mov     eax, ds:dword_4023B4
00401142 push    offset aWrong   ; "WRONG"
00401147 push    0Bh
00401149 pop     ecx
0040114A cdq
0040114B idiv    ecx
0040114D push    ds:lpText[edx*4] ; lpText
00401154 push    edi             ; hWnd
00401155 call    ds:MessageBoxA
0040115B inc     ds:dword_4023B4
00401161 cmp     ds:dword_4023B4, 0Bh
00401168 jnz     loc_4010C9

0040110C push    esi             ; uType
0040110D push    offset aPfft_   ; "PFFT."
00401112 push    offset aThatsNotAPrope ; "Thats not a proper serial.. Use a norma"...

00401131 push    offset aRight   ; "RIGHT"
00401136 push    offset aYouGotIt ; "You got it!"
0040113B jmp     short loc_401117
```

Moving backwards from the desired result, we start identifying the conditions that have to be met:

1. **[0x0040112F] jz short loc_40113D -** The Zero Flag must be zero for the program to take the correct path
2. **[0x0040112A] test eax, eax -** Since we require ZF to be zero, EAX must NOT be 0x00000000.
3. **[0x00401125] call sub_401000 -** This requires further investigation but we know that when it ends, EAX cannot be zero

Before diving into this unknown function let's see if anything of interest happens before it is reached.



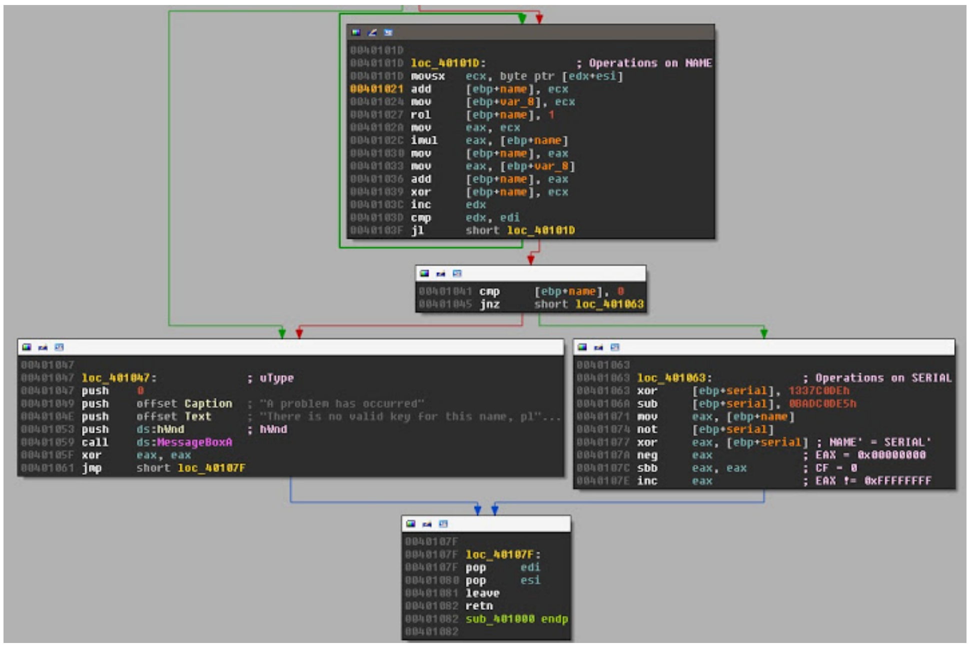
```
004010D2
004010D2 loc_4010D2:
004010D2 mov     edi, [ebp+hDlg]
004010D5 push    32h                 ; cchMax
004010D7 lea     eax, [ebp+String]
004010DA push    eax                 ; lpString
004010DB push    3E8h                ; nIDDlgItem
004010E0 push    edi                 ; hDlg
004010E1 call    ds:GetDlgItemTextA
004010E7 test    eax, eax
004010E9 jnz     short loc_4010F8
```

```
004010F8
004010F8 loc_4010F8:
004010F8 xor     esi, esi
004010FA push    esi                  ; bSigned
004010FB push    esi                  ; lpTranslated
004010FC push    3E9h                 ; nIDDlgItem
00401101 push    edi                  ; hDlg
00401102 call    ds:GetDlgItemInt
00401108 cmp     eax, esi
0040110A jnz     short loc_401120
```

```
00401120
00401120 loc_401120:                  ; serial
00401120 push    eax
00401121 lea     eax, [ebp+String]
00401124 push    eax                  ; lpString
00401125 call    sub_401000
0040112A test    eax, eax             ; ZF - 0 iff EAX !- 0x00000000
0040112C pop     ecx
0040112D pop     ecx
0040112E push    esi                  ; uType
0040112F jz      short loc_40113D ; ZF = 0 => DONE !
```

We notice 2 system calls: GetDlgItemTextA and GetDlgItemInt. According to MSDN, GetDlgItemTextA "retrieves the title or text associated with a control in a dialog box" while GetDlgItemInt "translates the text of a specified control in a dialog box into an integer value". These functions retrieve the values for **Name** and **Serial** from the textboxes. It seems that nothing of interest happens prior to function sub_401000, so let's focus on it.

The Main Fuction



```
0040101D
0040101D loc_40101D:                  ; Operations on NAME
0040101D movsx   ecx, byte ptr [edx+esi]
00401021 add     [ebp+name], ecx
00401024 mov     [ebp+var_8], ecx
00401027 rol     [ebp+name], 1
0040102A mov     eax, ecx
0040102C imul    eax, [ebp+name]
00401030 mov     [ebp+name], eax
00401033 mov     eax, [ebp+var_8]
00401036 add     [ebp+name], eax
00401039 xor     [ebp+name], ecx
0040103C inc     edx
0040103D cmp     edx, edi
0040103F jl      short loc_40101D
```

```
00401041 cmp     [ebp+name], 0
00401045 jnz     short loc_401063
```

```
00401047
00401047 loc_401047:                  ; uType
00401047 push    0
00401049 push    offset Caption     ; "A problem has occurred"
0040104E push    offset Text        ; "There is no valid key for this name, pl"...
00401053 push    ds:hWnd            ; hWnd
00401059 call    ds:MessageBoxA
0040105F xor     eax, eax
00401061 jmp     short loc_40107F
```

```
00401063
00401063 loc_401063:                  ; Operations on SERIAL
00401063 xor     [ebp+serial], 1337C0DEh
0040106A sub     [ebp+serial], 0BADC0DE5h
00401071 mov     eax, [ebp+name]
00401074 not     [ebp+serial]
00401077 xor     eax, [ebp+serial] ; NAME' = SERIAL'
0040107A neg     eax                ; EAX = 0x00000000
0040107C sbb     eax, eax           ; CF = 0
0040107E inc     eax                ; EAX != 0xFFFFFFFF
```

```
0040107F
0040107F loc_40107F:
0040107F pop     edi
00401080 pop     esi
00401081 leave
00401082 retn
00401082 sub_401000 endp
00401082
```

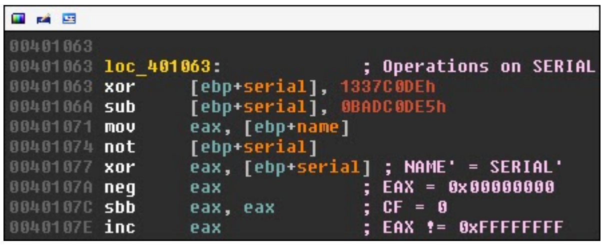
The function is manageable but we can still weed out some irrelevant parts so we can focus on the core functionality. Once again we start from the bottom:

1. **[0x0040107D] inc eax -** EAX cannot be 0xFFFFFFFF (Recall: EAX cannot be 0 when the function ends)
2. **[0x0040107E] sbb eax, eax -** Subtracting a number from itself gives zero, so CF has to be zero, else we get EAX = -1
3. **[0x0040107A] neg eax -** Only EAX = 0x00000000 will not set the Carry Flag
4. **[0x00401077] xor eax, [ebp+serial]** -To end up with a EAX = 0x00000000, EAX must be equal to [ebp+serial]

Let's stop there for the time being and move to the top. We skip the top most part (not shown in the image above) since it only checks if the length of **Name** is 0.

The subsequent section loads the name we've supplied and loops over each character, each time operating on [ebp+name]. By the end of the loop [ebp+name] holds the result. Let us name this end result **Name'**, since it is derived from **Name**. To keep things simple, and also since we'll be solving the binary for a specific input, we do not need to know exactly what the operations performed on **Name**, to produce **Name'**, are. The only thing we need to know is the value of **Name'**. By setting a break point at the end of the function, we get **Name'** = 0x01E55668.

With the value of **Name'** in hand, we can focus on the last part of the function. I've displayed this here for reference:



```
00401063
00401063 loc_401063:                ; Operations on SERIAL
00401063 xor     [ebp+serial], 1337C0DEh
0040106A sub     [ebp+serial], 0BADC0DE5h
00401071 mov     eax, [ebp+name]
00401074 not     [ebp+serial]
00401077 xor     eax, [ebp+serial] ; NAME' = SERIAL'
0040107A neg     eax               ; EAX = 0x00000000
0040107C sbb     eax, eax          ; CF = 0
0040107E inc     eax               ; EAX != 0xFFFFFFFF
```

As a reminder from the previous analysis, we require the XOR function at address 0x00401077 to result in a 0. The XOR's operands are **Name'** and **Serial'**, where **Serial'** is the resultant of **Serial** after some operations have been performed on it; we'll see what these operations are. This means that at this point, **SERIAL'** and **NAME'** have to be equal and we already know that **Name'** = 0x01E55668.

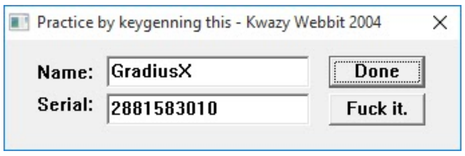
From the image above we can see that the operations on **Serial** are:

1. XOR Serial with 0x1337C0DE
2. SUB result with 0xBADC0DE5
3. NOT result

Combining these we end up with a formula with which we can work out what the **Serial** for our **Name** must be. Let's see how it's done.

```
                 NOT (( Serial ^ 0x1337C0DE ) - 0xBADC0DE5) = 0x01E55668

    =>                 ( Serial ^ 0x1337C0DE ) - 0xBADC0DE5 = 0xFE1AA997

    =>                                Serial ^ 0x1337C0DE = 0xB8F6B77C

    =>                                             Serial = 0xABC177A2

    =>                                             Serial = 2881583010
```

Combining everything together we try our luck again ...

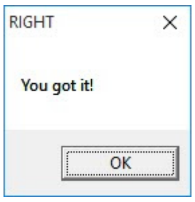


Practice by keygenning this - Kwazy Webbit 2004

Name: GradiusX

Serial: 2881583010

... and ..



RIGHT

You got it!

... we've done it !!

Conclusion

It was a long, bumpy road but at the end we've manage to find a **Name-Serial** pair that worked. As an exercise I suggest solving the keygen for a different **Name** or even for an arbitrary **Name**.

The plan for the next reverse engineering tutorial is to cover one of the Flare-on challenges so stay tuned .. じやあ、また

Posted by Francesco Mifsud at 02:37   No comments: G+1 Recommend this on Google