ISEL | DEETC

Programação em Sistemas Computacionais

2ª Série de Exercícios

Semestre de Inverno de 2009/2010

Autores:
30896 – Ricardo Canto
31401 – Nuno Cancelo
33595 – Nuno Sousa

Enunciado:

4. O programa fornecido pede 4 senhas de acesso ao utilizador. Pretende-se que descubra quais são as senhas correctas a introduzir, por análise do conteúdo do ficheiro fornecido e da sua execução, tendo sempre em conta que a introdução de uma senha errada tem consequências indesejáveis. Para obter o programa, no site uThoth da respectiva turma, escolha apenas o ficheiro com o número do seu grupo.

Resolução

Exercício 4:

Com o objectivo de encontrar as 4 senhas de acesso pedidas foi necessário analisaro código do programa em *C* e efectuar o debug ao programa *bomb* com o *insight.*

Após uma primeira análise é possível verificar que existe uma o ficheiro *bomb* prepara os códigos a pedir ao utilizador e o programa *bomb.c* se limita a pedir ao utilizador a chave para a fase respectiva e, dependendo da resposta sai do programa ou continua para a fase seguinte.

Iremos efectuar a análise das diversas fases individualmente.

FASE 1:

Na primeira fase o código é bastante simples. É chamada uma função que recebe a string introduzida como parâmetro:

```
push   ebp
mov    ebp,esp
push   0x804b060                    (2)
call   0x8048a20 <unscramble>       (1)
mov    DWORD PTR [esp],0x804b060    (3)
push   DWORD PTR [ebp+0x8]          (4)
call   0x80484d0 <strcmp@plt>       (5)
add    esp,0x8
test   eax,eax
sete   al                           (6)
movzx  eax,al
leave
ret
```

É chamada a função que se encontra em 0x8048a20 (1) passando como argumento a string que se encontra guardada em 0x804b060 (1&S2...|.9\1.a.`.sS7B0Qq.d.~.1S2F#.b.sTt.p.5S&T5.Z.) (2).

Esta função vai descodificar a string referida:

```
push   ebp
mov    ebp,esp
push   esi
push   ebx
mov    ebx,DWORD PTR [ebp+0x8]
mov    cl,BYTE PTR [ebx]
movsx  edx,cl
lea    esi,[ebx+edx]
mov    al,BYTE PTR [ebx+edx-0x1]
xor    al,BYTE PTR [esi]
mov    BYTE PTR [ebx],al
lea    eax,[edx-0x1]
```

```
test   eax,eax
jle    0x8048a54 <unscramble+52>
lea    edx,[ebx+eax]
movsx  ecx,cl
mov    eax,edx
sub    eax,ecx
mov    al,BYTE PTR [eax+ecx-0x1]
xor    BYTE PTR [edx],al
dec    edx
cmp    ebx,edx
jne    0x8048a45 <unscramble+37>
mov    BYTE PTR [esi],0x0
pop    ebx
pop    esi
leave
ret
```

O código do primeiro caracter da string vai servir de índice para o caracter que fica n posições à frente na mesma e depois, com uma série de manipulações de XOR entre dois caracteres utilizando sempre o código do primeiro caracter como índice a string que fica na memória acaba por ser a resposta esperada à primeira chave.

Na função anterior chama-se a função *strcmp* (5) da biblioteca *string.h* do *c* passando como parâmetros a string já decifrada (3) e a string introduzida (2). Se forem iguais coloca 1 em *EAX* e retorna à função anterior.

A chave para esta fase é: "**Agua mole em pedra dura tanto bate ate' que fura.**"

FASE 2:

Na segunda fase rapidamente se percebe que a palavra chave é um número uma vez que é chamada a função *readInteger*.

```
push   ebp
mov    ebp,esp
sub    esp,0x8
mov    DWORD PTR [ebp-0x8],0x0
lea    eax,[ebp-0x4]
push   eax
push   DWORD PTR [ebp+0x8]
call   0x8048994 <readInteger> /*transforma a string que recebeu em inteiro*/
add    esp,0x8
test   eax,eax
je     0x8048784 <phase2+68>
cmp    BYTE PTR [eax],0x0
jne    0x8048784 <phase2+68>
lea    eax,[ebp-0x8]
push   eax
push   DWORD PTR [ebp-0x4]
call   0x80488b0 <func>        /*função que vai transformar o inteiro introduzido*/
add    esp,0x8
mov    eax,ds:0x804b0e0        /*valor esperado a comparar como o resultado da função
                                 anterior*/
cmp    eax,DWORD PTR [ebp-0x8]
sete   al
movzx  eax,al
jmp    0x8048789 <phase2+73>
mov    eax,0x0
leave
ret
```

Após uma rápida análise ao código da função chamada para transformar o nosso input é fácil perceber que o número introduzido é invertido 4 bits (um caracter) de cada vez e o resultado final terá de ser o valor que se encontra em 0x804b0e0 (0xEA4).

```
push   ebp
mov    ebp,esp
push   ebx
mov    ebx,DWORD PTR [ebp+0xc]
mov    ecx,DWORD PTR [ebp+0x8]
mov    eax,DWORD PTR [ebx]
```

```
shl    eax,0x4
mov    edx,ecx
and    edx,0xf
add    eax,edx
mov    DWORD PTR [ebx],eax
cmp    ecx,0xf
jbe    0x80488dc <func+44>
push   ebx
mov    eax,ecx
shr    eax,0x4
push   eax
call   0x80488b0 <func>
add    esp,0x8
mov    ebx,DWORD PTR [ebp-0x4]
leave
ret
```

Como o resultado esperado é 0xEA4 então a palavra chave é 0x4AE.

A chave para esta fase é: **1198**.

FASE 3:

A análise da terceira fase já requere mais atenção:

```
       push   ebp
       mov    ebp,esp
       push   edi
       push   esi
       push   ebx
       sub    esp,0x10
       mov    esi,DWORD PTR [ebp+0x8]
       push   esi
       call   0x8048490 <strlen@plt>
       add    esp,0x4
       sub    eax,0x4                      (1)
       cmp    eax,0xb                      (1)
       ja     0x80487f9 <phase3+109>
       movsx  eax,BYTE PTR [esi]
       mov    ebx,DWORD PTR ds:0x804b0e4
       mov    ecx,0x0
       lea    edi,[ebp-0x1c]
<phase3+46>:
       mov    edx,eax
       lea    eax,[eax-0x41]               (2)
       cmp    eax,0xf                      (2)
       ja     0x80487f9 <phase3+109>
       mov    eax,edx
       and    eax,0xf
       mov    al,BYTE PTR [ebx+eax]
       mov    BYTE PTR [edi+ecx],al
       inc    ecx
       movsx  eax,BYTE PTR [esi+ecx]
       test   eax,eax
       jne    0x80487ba <phase3+46>
       mov    BYTE PTR [ebp+ecx-0x1c],0x0
       push   DWORD PTR ds:0x804b0e8
       lea    eax,[ebp-0x1c]
       push   eax
       call   0x80484d0 <strcmp@plt>       (3)
       add    esp,0x8
       test   eax,eax
       sete   al
       movzx  eax,al
       jmp    0x80487fe <phase3+114>
<phase3+109>:
       mov    eax,0x0
<phase3+114>:
       lea    esp,[ebp-0xc]
       pop    ebx
       pop    esi
       pop    edi
       leave
       ret
```

Em (1) é verificado logo o tamanho da resposta. Se esta tiver mais do que 15 (comprimento da string menos 4 tem de ser inferior ou igual a 0xb (11)) caracteres sai imediatamente.

Em (2) verifica-se que o código ASCII dos caracteres tem de estar entre 0x41 (65) e 0x50 (80) ou seja entre 'A' e 'P'. Isto porque retirando 0x41 ao código do caracter este tem de ser menor ou igual a 0xf (15).

Aos caracteres introduzidos é efectuado um AND com 0xf, ou seja, aproveitam-se apenas os 4 bits menos significativos, sendo que o resultado ficará sempre entre 0 e 15 (devido às restrições anteriores). Esses valores irão servir de índice para ir buscar os caracteres correctos à string que se encontra guardada em 0x804b0e4: {tblofeocadpinrcr}. O objectivo final (3) é comparar a string com "correcto" logo existem várias hipóteses diferentes para o código.

tblofeocadpinrcr

correcto
GCMMEGPC
NFOO N F

Uma das chaves possíveis para esta fase é: **GFMOENPC**.

FASE 4:

A 4ª fase foi bastante mais difícil de descobrir tendo em conta que as restrições são muito maiores.

```
        push   ebp
        mov    ebp,esp
        push   edi
        push   esi
        push   ebx
        sub    esp,0x24                      (1)
        mov    DWORD PTR [ebp-0x10],0x7
        lea    eax,[ebp-0x10]
        push   eax
        lea    eax,[ebp-0x2c]
        push   eax                           (2..)   /*0x7*/
        push   DWORD PTR [ebp+0x8]                   /*valor passado*/
        call   0x80489d0 <readNIntegers>     (..2)   /*ler EAX números*/
        add    esp,0xc
        cmp    DWORD PTR [ebp-0x10],0x0      (3)
        jne    0x80488a5 <phase4+157>
        cmp    BYTE PTR [eax],0x0
        jne    0x80488a5 <phase4+157>
        mov    ebx,DWORD PTR ds:0x804b0ec    (4a)
        mov    ecx,DWORD PTR ds:0x804b0f0    (4b)
        mov    edx,DWORD PTR ds:0x804b0f4    (4c)
        lea    esi,[ebp-0x28]
        mov    edi,esi
        lea    eax,[ebp-0x10]
        mov    DWORD PTR [ebp-0x30],eax
<phase4+75>:
        mov    eax,DWORD PTR [edi]           (5..)
        cmp    eax,ebx
        je     0x8048861 <phase4+89>
        cmp    eax,ecx
        je     0x8048861 <phase4+89>
        cmp    eax,edx
        jne    0x80488a5 <phase4+157>
<phase4+89>:
        add    edi,0x8
        cmp    edi,DWORD PTR [ebp-0x30]
        jne    0x8048853 <phase4+75>         (..5)
        mov    eax,DWORD PTR [ebp-0x2c]
<phase4+100>:
        mov    ebx,DWORD PTR [esi]
        cmp    DWORD PTR [ebx*8+0x804b110],0x0      (6)
        jne    0x80488a5 <phase4+157>
        push   DWORD PTR [esi+0x4]
        push   eax
        call   DWORD PTR [ebx*8+0x804b10c]          (7)
        pop    edx
```

```
        pop    ecx
        mov    DWORD PTR [ebx*8+0x804b110],0x1      (6a)
        add    esi,0x8
        cmp    esi,edi
        jne    0x804886c <phase4+100>
        cmp    eax,DWORD PTR ds:0x804b0f8           (8)
        sete   al
        movzx  eax,al
        jmp    0x80488a7 <phase4+159>
<phase4+157>:
        xor    eax,eax
<phase4+159>:
        lea    esp,[ebp-0xc]
        pop    ebx
        pop    esi
        pop    edi
        leave
        ret
```

Numa primeira análise é reservado espaço (1) no *stack* para variáveis locais. No bloco indicado por (2) e pela análise da função *readNIntegers* percebemos que é necessário introduzir 7 números distintos que vão ser guardados no *stack* nas posições situadas entre [ebp -0x2c] e [ebp-0x14]. Por (3) percebe-se que temos de introduzir 7 e apenas 7 números. Na análise da função *readNIntegers* verifica-se também estes números têm de ser separados por um espaço (código ASCII 0x20) ou por um tab (código ASCII 0x9):

```
<readNIntegers+42>:
        inc    ecx
<readNIntegers+43>:
        mov    dl,BYTE PTR [ecx]
        cmp    dl,0x20
        je     0x80489fa <readNIntegers+42>
        cmp    dl,0x9
        je     0x80489fa <readNIntegers+42>
        test   dl,dl
        je     0x8048a14 <readNIntegers+68>
```

Por (4) e por (5) verificamos que os argumentos nas posições 2, 4 e 6 têm de ser iguais a 0x804b0ec (valor 6), 0x804b0f0 (valor 2) e 0x804b0f4 (valor 7).

Em (6) e (6a) torna-se claro que os números das posições 2, 4 e 6 não se podem repetir, ou seja, os argumentos com valor 6, 2 e 7 só podem ser usados uma vez cada.

Em (7) é chamada uma função que depende do valor do argumento nas posições pares dos números introduzidos. A primeira função usa como argumentos os números das posições 1 e 3, a segunda usa o resultado da primeira juntamente com o número na posição 5 e a terceira o resultado da anterior com o número da posição 7.

As três funções são (sendo *a* e *b* os argumentos genéricos):

parâmetro 2:

```
push   ebp
mov    ebp,esp
mov    eax,DWORD PTR [ebp+0xc]
imul   eax,DWORD PTR [ebp+0x8]
leave
ret
```

Devolve o resultado de (a*b).

parâmetro 6:

```
push   ebp
mov    ebp,esp
mov    eax,DWORD PTR [ebp+0xc]
xor    eax,DWORD PTR [ebp+0x8]
leave
ret
```

Devolve o resultado de (a XOR b).
parâmetro 7:

```
push                                                    ebp
mov                                                   ebp,esp
mov                                  edx,DWORD          PTR          [ebp+0x8]
```

```
mov                                     ecx,DWORD           PTR           [ebp+0xc]
mov                                                                         eax,ecx
imul                                                                        eax,edx
add                                                                         ecx,edx
cdq
idiv                                                                            ecx
leave
ret
```

Devolve o resultado de ((a * b)/(a+b)).

Para obtermos a chave só falta saber qual o resultado esperado. Este encontra-se em 0x804b0f8 (8) e é 0x25, ou seja, 37.

Uma vez que podemos escolher a ordem das funções a aplicar a função '2' nunca pode ser a última uma vez que 37 é um número primo.

Assim, escolhemos como última função a '7':

37 – 00100101

que pode ser decomposto (XOR) em:

32 – 00100000

05 – 00000101

Assim, o último argumento pode ser 5, o da posição 6 tem de ser 7.

Se o da posição 4 for 2 então o da posição 2 terá de ser 6. Se usarmos 9 na posição 1 e 3 então temos 4 como resultado da função '6' e o da posição 5 terá de ser 8 para que 8*4=32.

Uma das chaves possíveis para esta fase é: **9 7 9 2 8 6 5**.

```
Tem como objectivo descobrir a senha associada a cada uma das 4 fases
Boa sorte!
Senha 1: Agua mole em pedra dura tanto bate ate' que fura.
OK, esta era simples! Que tal a seguinte?
Senha 2: 1198
Bem conseguido! E agora?
Senha 3: GFMOENPC
Boa! Mais uma?
Senha 4: 9 7 9 2 8 6 5
Parabens! Resolveu todas as fases!
```

Este exercício ajudou-nos bastante a perceber melhor (e na prática) a passagem de parâmetros entre funções em Assembly IA-32 e alguns truques usados para aumentar a velocidade de execução dos programas (o uso do LEA em vez do MOV, o do XOR a, a em vez do MOV a, 0 , etc.).