Crackmes.de – KeygenMe #2 by Lesco

Johannes

January 29, 2015

The crackme KeygenMe #2 by Lesco has been published August 12, 2006. It is rated at *"5 - Professional problem to solve"*. The crackme is written in C/C++ and runs on Windows. The description reads:

> This time you'll have to solve a little math problem. I didn't use any protectors and anti-debug stuff, since the focus of the crackme is the algorithm. The goal is to write a working keygen+explanation of the crackme.There's only one rule: Don't patch!

The first two sections of this solution show how to reverse engineer the two main parts of the code. The crackme does not use any anti-debug or anti-disassembly techniques and it is trivial to reverse considering the difficulty rating. The third section then shows how to solve the *"little math problem"*, and how to build a key generator.

At the heart of the crackme are two subroutines which I named `validate` and `evaluate`, the latter being called by `validate`. The `evaluate` subroutine transforms the serial into three values; the `validate` does the same for the name and checks if the values correspond to the result of `evaluate`.

Reversing "validate"

The `validate` subroutine does essentially five things:

1. It checks the length of the name.
2. It converts the name to a 32bit hash.
3. It chops the hash value into two vectors of three values each.
4. It calls `evaluate` for all elements of the first vector.
5. It compares the result of `evaluate` to the second vector.

Checking the Length of the Name

The subroutine `validate` gets the name as the first parameter, and the serial as the second. It returns 0 if the serial is invalid, and 1 if the serial is valid. The start of `validate` disassembles to these lines:

```
00401320 validate proc near
00401320
00401320 var_3C= qword ptr -3Ch
00401320 y= dword ptr -24h
00401320 x= dword ptr -18h
00401320 result= qword ptr -0Ch
00401320 var_4= dword ptr -4
00401320 name= dword ptr  8
```

```
00401320 serial= dword ptr  0Ch
00401320
00401320 push    ebp
00401321 mov     ebp, esp
00401323 sub     esp, 24h
00401326 push    ebx
00401327 mov     ebx, [ebp+name]
0040132A push    esi
0040132B push    edi
0040132C mov     edi, ebx
0040132E or      ecx, 0FFFFFFFFh
00401331 xor     eax, eax
00401333 repne scasb
00401335 not     ecx
00401337 dec     ecx
00401338 mov     [ebp+name], ecx
0040133B lea     ecx, [ebp+name]
00401341 mov     eax, [ecx]
00401343 imul    eax, 0DEADh
00401349 lea     esi, [ebp+var_4]
0040134F mov     [esi], eax
00401351 mov     eax, [ebp+var_4]
00401354 cmp     eax, 29C07h
00401359 jnb     short loc_401364
0040135B pop     edi
0040135C pop     esi
0040135D xor     al, al
0040135F pop     ebx
00401360 mov     esp, ebp
00401362 pop     ebp
00401363 retn
00401364 loc_401364:
00401364 cmp     eax, 116584h
00401369 jbe     short loc_401374
0040136B pop     edi
0040136C pop     esi
0040136D xor     al, al
0040136F pop     ebx
00401370 mov     esp, ebp
00401372 pop     ebp
00401373 retn
```

The snippet checks:

```
if 57005 * strlen(name) < 0x29C07 or 57005 * strlen(name) > 0x116584:
    return 0 # you fail
else
    # continue
```

which of course is

```
if 3 <= strlen(name) <= 20:
    # continue
else
    return 0 # you fail
```

Hashing the Name

Next follows this loop:

```
00401374 loc_401374:
00401374 mov     esi, [ebp+name]
00401377 xor     edx, edx
00401379 test    esi, esi
0040137B mov     eax, 0DEADCA7h
00401380 jle     short loc_4013B0
00401382
00401382 loc_401382:
00401382 mov     cl, [edx+ebx]
00401385 and     ecx, 0FFh
0040138B mov     edi, ecx
0040138D mov     [ebp+var_4], ecx
00401390 imul    edi, 0F28437h
00401396 xor     edi, eax
00401398 mov     eax, ecx
0040139A imul    ecx, 0D23664h
004013A0 shr     eax, 2
004013A3 add     edi, eax
004013A5 not     edi
004013A7 sub     edi, ecx
004013A9 inc     edx
004013AA cmp     edx, esi
004013AC mov     eax, edi
004013AE jl      short loc_401382
```

These lines compute a 32bit hash value of the name:

```
h = 0xdeadca7
for n in name:
    h = ( ~((n >> 2) + (h ^ 0xf28437*n)) - 0xd23664*n ) & 0xFFFFFFFF
```

The computation looks reasonably complicated, so let's just assume we have to deal with random 32bit numbers as input.

Chopping the Hash into 6 values

The name hash (in `eax`) is then split into 6 values, which are stored in two vectors of three elements each. I named the first vector `x`, and the second `y`:

```
004013B0 loc_4013B0:
004013B0 xor     ebx, ebx
004013B2 xor     esi, esi
004013B4
004013B4 loc_4013B4:
004013B4 mov     ecx, eax
004013B6 and     ecx, 0Fh
004013B9 shr     eax, 4
004013BC inc     ecx
```

```
004013BD test    al, 1
004013BF mov     [ebp+esi+x], ecx
004013C3 jz      short loc_4013CD
004013C5 mov     edx, ecx
004013C7 neg     edx
004013C9 mov     [ebp+esi+x], edx
004013CD
004013CD loc_4013CD:
004013CD shr     eax, 1
004013CF mov     ecx, eax
004013D1 and     ecx, 0Fh
004013D4 shr     eax, 4
004013D7 inc     ecx
004013D8 test    al, 1
004013DA mov     [ebp+esi+y], ecx
004013DE jz      short loc_4013E8
004013E0 mov     edx, ecx
004013E2 neg     edx
004013E4 mov     [ebp+esi+y], edx
004013E8
004013E8 loc_4013E8:
004013E8 shr     eax, 1
004013EA test    esi, esi
004013EC jle     short loc_401404
004013EE lea     ecx, [ebp+x]
004013F1 mov     edi, ebx
004013F3
004013F3 loc_4013F3:
004013F3 mov     edx, [ecx]
004013F5 cmp     edx, [ebp+esi+x]
004013F9 jnz     short loc_4013FE
004013FB inc     edx
004013FC mov     [ecx], edx
004013FE
004013FE loc_4013FE:
004013FE add     ecx, 4
00401401 dec     edi
00401402 jnz     short loc_4013F3
00401404
00401404 loc_401404:
00401404 add     esi, 4
00401407 inc     ebx
00401408 cmp     esi, 0Ch
0040140B jl      short loc_4013B4
```

This code uses 5 bits per value. The first four bits are used for the absolute value; The fifth and most significant bit is the sign. The following image illustrates which parts of the hash will be used for which vector element:

```
. 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
.|S |     y_3   |S |     x_3    |S |     y_2    |S |     x_2    |S |     y_1    |S |     x_1    |
-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
. ^ sign y_3     ^ sign x_3     ^ sign y_2     ^ sign x_2     ^ sign y_1     ^ sign x_1
```

The code also adds 1 to each absolute value, and it makes sure all three values `x_1`, `x_2`, and `x_3` are different. Here is the pseudo-code of the snippet:

```
h = hash_name(name)
x, y = [], []
for i in range(6):
    val = (h & 0xF) + 1
    if h & 0x10:
        val *= -1;
    if not i%2:
        x.append(val)
    else:
        y.append(val)

    if i%2:
        for j in range(i//2):
            if x[j] == x[i//2]:
                x[j] += 1
    h = h >> 5;
```

Calling "evaluate"

After the crackme has built the two vectors x and y, it calls the subroutine `evaluate` three times for each of the three elements of x:

```
0040140D mov     edi, [ebp+serial]
00401410 xor     esi, esi
00401412
00401412 loc_401412:
00401412 fild    [ebp+esi+x]
00401416 lea     eax, [ebp+f]
00401419 push    eax              ; f
0040141A sub     esp, 8
0040141D fstp    [esp+3Ch+var_3C] ; x_i
00401420 push    edi              ; serial
00401421 call    evaluate
....
```

from this snippet we see the function prototype of `evaluate`:

```
int evaluate(char* serial, double x_i, double* f)
```

where `serial` is a pointer to the entered serial string, `x_i` is the *ith* value of vector x, and f is a pointer to a double that will receive the result of `evaluate`. The function returns 0 if there was an error evaluating the serial, and non-zero value otherwise.

Checking the Result of "evaluate"

For each of the three calls to `evaluate`, the crackme checks if `evaluate` was successful and returns 0 otherwise.

```
00401421 call    evaluate
00401426 add     esp, 10h
00401429 test    al, al
0040142B jz      short fail
```

If on the other hand, the return value of `evaluate` is non-zero, the result in `[ebp+f]` is compared to the values in vector *y*:

```
0040142D fild    [ebp+esi+y]
00401431 fld     [ebp+f]
00401434 fld     st(1)
00401436 fadd    ds:c_0_01
0040143C fld     st(1)
0040143E fcompp
00401440 fnstsw  ax
00401442 test    ah, 'A'
00401445 jz      short fail2
00401447 fxch    st(1)
00401449 fsub    ds:c_0_01
0040144F fxch    st(1)
00401451 fcompp
00401453 fnstsw  ax
00401455 test    ah, 1
00401458 jnz     short fail
0040145A add     esi, 4
0040145D cmp     esi, 0Ch
00401460 jl      short loc_401412
```

The result of `evaluate` only needs to be with +- 0.01 of *yi* to be accepted. Let *e* denote the subroutine evaluate, and let *s* be the serial. Also, let *x = (x1, x2, x3)* denote the vector *x*, and *y = (y1, y2, y3)* the vector *y*. Then the `validate` routine checks the following math equation

$$\forall_{i \in 1,2,3} \, |e(s, x_i) - y_i| \leq 0.01$$

Reversing "Evaluate"

So all boils down to the magic function *e*, represented by the subroutine `evaluate`. The disassembly for `evaluate` is rather long, featuring a lot of conditionals. Fortunately, there are many tell-tale comparisons to constants that show us the meaning of the subroutine. I gradually reversed `evaluate` based on the constants.

Numbers in Scientific Form

The first interesting comparisons compare characters of the serial to "0", "9", "-", ".", "e" and "E". We find these comparisons at various locations throughout `evaluate`, for example:

```
00401090 mov     al, [esi+ecx]
00401093 cmp     al, '9'
00401095 jg      short loc_40109B
00401097 cmp     al, '0'
00401099 jge     short loc_4010AB
0040109B
```

```
0040109B loc_40109B:
0040109B cmp     al, '.'
0040109D jz      short loc_4010AB
0040109F cmp     al, 'e'
004010A1 jz      short loc_4010AB
004010A3 cmp     al, 'E'
004010A5 jz      short loc_4010AB
004010A7 cmp     al, '-'
004010A9 jnz     short loc_4010C2
```

These character constants hint at scientific numbers: the routine `evaluate` interprets the serial string as a scientific number. We can easily test this hypothesis by entering `-1.2345e2` as the serial and checking the result of top of the FPU register stack after this instruction:

```
00401431 fld     [ebp+f]
```

The value of `ST0` should be *-123.45*. So `evaluate` correctly parses numbers in scientific form. But `evaluate` has more features.

Mathematical Expressions

Towards the end of `evaluate` we find the following interesting switch statement:

```
00401225 cmp     eax, 3Ch                         ; switch 61 cases
00401228 ja      loc_4012B9                       ; jumptable 00401236 default case
0040122E xor     ecx, ecx
00401230 mov     cl, ds:byte_4012DC[eax]
00401236 jmp     ds:off_4012C4[ecx*4]             ; switch jump
0040123D ; ---------------------------------------------------------------------------
0040123D
0040123D loc_40123D:                              ; CODE XREF: evaluate+236j
0040123D                                          ; DATA XREF: .text:off_4012C4o
0040123D fld     [esp+30h+var_8]                  ; jumptable 00401236 case 1
00401241 fcomp   ds:const_2_0
00401247 fnstsw  ax
00401249 test    ah, 41h
0040124C jz      short loc_4012B9                 ; jumptable 00401236 default case
0040124E fld     [esp+30h+var_10]
00401252 fld     [esp+30h+var_8]
00401256 call    __CIpow
0040125B jmp     short loc_401283
0040125D ; ---------------------------------------------------------------------------
0040125D
0040125D loc_40125D:                              ; CODE XREF: evaluate+236j
0040125D                                          ; DATA XREF: .text:off_4012C4o
0040125D fld     [esp+30h+var_8]                  ; jumptable 00401236 case 0
00401261 fadd    [esp+30h+var_10]
00401265 jmp     short loc_401283
00401267 ; ---------------------------------------------------------------------------
00401267
00401267 loc_401267:                              ; CODE XREF: evaluate+236j
00401267                                          ; DATA XREF: .text:off_4012C4o
00401267 fld     [esp+30h+var_10]                 ; jumptable 00401236 case 60
```

```
0040126B fsub    [esp+30h+var_8]
0040126F jmp     short loc_401283
00401271 ; ---------------------------------------------------------------------------
00401271
00401271 loc_401271:                             ; CODE XREF: evaluate+236j
00401271                                         ; DATA XREF: .text:off_4012C4o
00401271 fld     [esp+30h+var_8]                 ; jumptable 00401236 case 28
00401275 fmul    [esp+30h+var_10]
00401279 jmp     short loc_401283
0040127B ; ---------------------------------------------------------------------------
0040127B
0040127B loc_40127B:                             ; CODE XREF: evaluate+236j
0040127B                                         ; DATA XREF: .text:off_4012C4o
0040127B fld     [esp+30h+var_10]                ; jumptable 00401236 case 3
0040127F fdiv    [esp+30h+var_8]
```

We see the arithmetic instructions **fadd**, **fsub**, **fmul**, and **fdiv**, as well as a call to the library function **pow**. In C pseudo code the switch statement is:

```
switch ( operation )
    {
        case '$':
            if ( op2 > 2.0 )
                return 0;
            f = pow(op1, op2);
            break;
        case '#':
            f = op1 + op2;
            break;
        case '_':
            f = op1 - op2;
            break;
        case '?':
            f = op1 * op2;
            break;
        case '&':
            f = op1 / op2;
            break;
        default:
        return 0;
    }
```

So `evaluate` can handle five math operations. The symbols it uses are non standard, the following table summarizes them:

Symbol	Meaning
`$`	Exponentiation (limited!)
`#`	Addition
`_`	Subtraction
`?`	Multiplication
`&`	Division

The `evaluate` routine won't follow the usual order of operations, e.g., multiplication before addition, but instead just evaluates the serial left to right. For example:

```
3#2&7
```

is evaluated to $(3+2)/7 = 0.7142$. So how do we group expression?

Brackets

At multiple points we see comparisons to "[" and "]":

```
004010F3 cmp     al, '['
004010F5 jnz     short loc_4010FA
004010F7 inc     ecx
004010F8 jmp     short loc_4010FF
004010FA ; -------------------------------------
004010FA
004010FA loc_4010FA:
004010FA cmp     al, ']'
```

Also, there are recursive calls to `evaluate`:

```
0040114F mov     edx, [ebp+high_32]
00401152 mov     eax, [ebp+low_32]
00401155 lea     ecx, [esp+30h+var_8]
00401159 push    ecx
0040115A push    edx
0040115B push    eax
0040115C push    edi
0040115D call    evaluate
```

This indicates that `evaluate` supports grouping of expressions inside "[" and "]". So

```
7?[3_1]
```

is evaluated as $7(3-1) = 14$.

So far we know that `evaluate` can calculate mathematical expression. Numbers can be written in scientific form, expressions can be grouped with brackets, and five operations are supported. But why does `evaluate` also take *xi* as an argument?

Variable x

There is one more interesting comparison to a constant that we have not covered yet:

```
004011B0 cmp     al, 'X'
```

All instances of "X" inside the serial will be replaced with the value of *xi*. For example,

```
serial: X$2
x: (1,2,3)
```

will be evaluated to 1, 4, and 9. This concludes the functionalities of `evaluate`. To summarize, the routine interprets a mathematical expression represented by the serial. The next section shows how to chose a valid mathematical expression for a given name.

Solving the Little Math Problem

Let *fs* denote the function represented by the serial *s*. Form the previous section we know that *fs* can contain multiplication, division, addition, subtraction and exponentiation. We also know that we need:

$$f_s(x) = \begin{cases} y_1 \pm 0.01 & x = x_1 \\ y_2 \pm 0.01 & x = x_2 \\ y_3 \pm 0.01 & x = x_3 \\ ? & \text{otherwise} \end{cases}$$

where "?" denotes don't care — the function *f* is only ever evaluated for *x1*, *x2*, and *x3*. So how do we get the function *fs* to evaluate to *y1* when $x = x1$, to *y2* when $x = x2$ and to *y3* when $x = x3$? What we need is an indicator function *I*— a function that becomes 1 when x has a certain value and 0 otherwise:

$$I_y(x) = \begin{cases} 1 & x = y \\ 0 & \text{otherwise} \end{cases}$$

Given such an indicator function *I*, we can easily build our function *f*:

$$f(x) := I_{x_1}(x) \cdot y_1 + I_{x_2}(x) \cdot y_2 + I_{x_3}(x) \cdot y_3$$

The indicator function that came to my mind is:

$$I_y(x) := 0^{|x-y|}$$

It works because 00 is 1 by definition, and 0k is 0 for any non-zero positive value *k*. Because we don't have `abs` availabe, I went with squaring the difference:

$$I_y(x) := 0^{(x-y)^2}$$

Unfortunately, that still won't quite work because of a limitation of the exponentiation routine in `evaluate` that I did not mention before:

```
0040123D fld     [esp+30h+var_8]                 ; jumptable 00401236 case 1
00401241 fcomp   ds:const_2_0
00401247 fnstsw  ax
00401249 test    ah, 41h
0040124C jz      short loc_4012B9                ; jumptable 00401236 default case
0040124E fld     [esp+30h+var_10]
00401252 fld     [esp+30h+var_8]
00401256 call    __CIpow
```

which is

```
if ( op2 > 2.0 )
    return 0;
f = pow(op1, op2);
break;
```

So all exponents greater than 2 will cause `evaluate` to fail. However, we can easily fix this: from reversing the code we know that all *xi* and *yi* are between -16 and 18. Therefore

$$(x_i - y_i)^2 < 40^2 = 1600$$

So by dividing our exponent by 800, we can guarantee it is smaller than 2 without otherwise affecting our indicator function. Our final version therefore is:

$$I_y(x) := 0^{\frac{(x-y)^2}{800}}$$

All we need to do now is translate

$$f(x) := 0^{\frac{(x_1-y_1)^2}{800}} \cdot y_1 + 0^{\frac{(x_2-y_2)^2}{800}} \cdot y_2 + 0^{\frac{(x_3-y_3)^2}{800}} \cdot y_3$$

to a serial string. Just make sure to set enough brackets; negative numbers for instance need to be surrounded by brackets - otherwise the minus sign will be interpreted as subtraction. My code also changes double negatives to plus, so instead of *3-(-4)* I calculate *3+4*. The following Python code generates serial strings for arbitrary names:

```
import argparse

def hash_name(name):
    h = 0xdeadca7
    for n in [ord(x) for x in name]:
        h = ( ~((n >> 2) + (h ^ 0xf28437*n)) - 0xd23664*n ) & 0xFFFFFFFF
    return h

def name_to_values(name):
    h = hash_name(name)
    x, y = [], []
    for i in range(6):
        val = (h & 0xF) + 1
        if h & 0x10:
            val *= -1;
        if not i%2:
            x.append(val)
        else:
            y.append(val)

        if i%2:
            for j in range(i//2):
                if x[j] == x[i//2]:
                    x[j] += 1
        h = h >> 5;

    return x, y

def keygen(name):
    if not 3 <= len(name) <= 20:
        return "name needs to be 3 to 20 characters long"
    x, y = name_to_values(name)
    serial_comp = []
```

```
    for xx, yy in zip(x, y):
        sign = "_" if xx >= 0 else "#"
        serial_comp.append("[0$[X{}{}$2&800]?[{}]]".format(sign,abs(xx), yy))
    return "#".join(serial_comp)

if __name__=="__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("name")
    args = parser.parse_args()
    print(keygen(args.name))
```

For example:

```
$ python keygen.py deadcat7
[0$[X#8$2&800]?[-5]]#[0$[X_6$2&800]?[1]]#[0$[X_5$2&800]?[2]]
```