RootMe #1 by waganono

1. **Introduction.**

This is a quite interesting crackme, with some crypto and basic math. There isn't much of a code-playing involved. Most of the time you will spend identifying procedures and then solving the equations, most likely on paper.

The tools I used:

- IDA (with FindCrypt plugin)
- SAGE (open source equivalent of Mathematica)
- Visual Studio (coding the keymaker)
- Python (some simple scripts)
- HIEW (for patching annoyances)
- Some paper and pen to deal with math behind it

2. **First glance.**

When we run the crackme, we notice two things right away. First is that the window was marked as always-on-top, God i hate it when it happens. The other thing easily noticeable is that the mod playing in the background behaves weird when we debug the prog. If we're at it, you prolly noticed that every debug session gives you different image base. Well, that's because this file has ASLR turned on by default. Ok, let's deal with it one by one. There's nothing fancy here, just patch those areas:

ASLR

```
00148: 00 00 00 00 02 00 40 81 |......@  -> 00148: 00 00 00 00 02 00 00 81 |.......
```

Always-on-top

```
00C10: 99 2B C2 D1 F8 50 6A 00 |™+ÂŃřPj. -> 00C10: 99 2B C2 D1 F8 50 6A FF |™+ÂŃřPj
```

Turning off the music

```
00CF8: 57 57 6A 68 A3 20 DA 42 |WWjhŁ ÚB -> 00CF8: 90 90 90 90 A3 20 DA 42 |Ł ÚB
00D00: 00 E8 CE 5E 01 00 6A 05 |.čÎ^..j.    00D00: 00 90 90 90 90 90 6A 05 |.j.
```

After those modifications, we see that there's no music playing, crackme isn't always-on-top window and we can easily hide it and of course when we run it under debugger it always has the same image base. The last point can also be achieved by stripping relocations from the file as that will force the program to run with its normal image base.

Finally we can load our target to IDA and see what's there.

3. **Algoritmus imposibilus (well, not quite)**

After loading the crackme into IDA i ran FindCrypt plugin (link should be available at the bottom of this paper) to identify most common crypto routines. It found Whirlpool and nothing else. But PEiD on the other hand, found also big number. That's a start. I will not beat around the bush, the most tiresome part of this crackme, was definetaly recognition of functions. There are few things you can do about it. If you have a slight idea what library it is, you can use signatures. In case of big complex libraries, like NTL, GMP or Crypto++ the chances of recognition are slim. The next logical step would be blackboxing, in my (and many other people's) opinion it is the most efficient method of recognising what does this function does. Especially if you use-pre set values and you know the results (if in put is 2,3 and 5, and output is 3, you can be almost sure that it is powmod, to ensure redo the test with different values). The last one that instantly sprint into mind, is only viable if you know the library used, you decompile the library in IDA

and manually compare procedures. This however takes really long time and thus is not really efficient. I will spare you this grief and here is the listing of the most important functions that are used in the crackme:

```
big_sub_int           00403730
big_size_in_base      00403C60
big_shift_right_int   004037F0
big_set_int           00403AF0
big_powmod_int        00404960
big_mul_int           00403F00
big_mod               00403050
big_legendre          00404040
big_init              00403C30
big_in_base           00404DC0
big_import            00404540
big_divide_qr_int     00403B30
big_divide_q_2exp     00403AD0
big_div_q             00402FC0
big_destroy           00403B10
big_compare           004047D0
big_cmp_int           00403BF0
big_and               00403230
big_add_int           00403E40
```

With this knowledge, we can start right off with the code.

We easily find the checking routine that is placed after two GetDlgItemTextA. Inside, we see (pseudocode interpretation):

```
check_serial(serial, name)
{
      a = big_init();
      b = big_init();
      c = big_init();
      d = big_init();
      big_in_base(n, modulus, 0x10);
      FirstCheck(serial,a,n,c,d);
      big_powmod_int(a,a,2,modulus);
      SecondCheck(name,c,d,modulus,b);
      r = big_compare(a,b);
      big_destroy(n);
      big_destroy(a);
      big_destroy(b);
      big_destroy(c);
      big_destroy(d);
      return r;
}
```

From this we see, that FirstCheck does something with our serial, returns a big number 'a' and then squares it modulo n, then modifies our name with numbers c and d (that were modified inside FirstCheck) and then the result is compared. This is generally speaking Rabin signature algorithm. Although this knowledge is not needed nor required to solve this crackme. Only thing needed is knowledge of modular arithmetic and an algorithm how to compute square roots modulo number. All of them will be explained later on.

Let's crack on and see what's insdie FirstCheck function.

This part can be divided into few steps. So let's do it one by one, shall we?

3a. **Base36 conversion**

First part of the check present itself like this:

```
.text:00401078                 mov     ebx, offset aQzm9gyldr4bks6 ; "QZM9GYLDR4BKS
.text:0040107D
.text:0040107D loc_40107D:                             ; CODE XREF: FirstCheck+6A↓j
.text:0040107D                 mov     ecx, [ebp+lpString]
.text:00401080                 mov     eax, [ebp+var_4]
.text:00401083                 movsx   eax, byte ptr [eax+ecx]
.text:00401087                 push    eax             ; int
.text:00401088                 push    ebx             ; char *
.text:00401089                 call    _strchr
.text:0040108E                 mov     edi, eax
.text:00401090                 pop     ecx
.text:00401091                 pop     ecx
.text:00401092                 test    edi, edi
.text:00401094                 jz      bad_boy
.text:0040109A                 push    36
.text:0040109C                 push    esi
.text:0040109D                 push    esi
.text:0040109E                 call    big_mul_int
.text:004010A3                 sub     edi, ebx
.text:004010A5                 push    edi
.text:004010A6                 push    esi
.text:004010A7                 push    esi
.text:004010A8                 call    big_add_int
.text:004010AD                 add     esp, 18h
.text:004010B0                 inc     [ebp+var_4]
.text:004010B3                 mov     eax, [ebp+var_4]
.text:004010B6                 cmp     eax, [ebp+var_14]
.text:004010B9                 jl      short loc_40107D
```

So as we see, first code checks if the i-th character of our serial is present inside the charset, if it does, it multiplies our big number by 36 and then adds to it position of that character inside the charset. Base conversion, plain and simple. So as a result of that part we have a big number which is a direct representation (in a different base) of our entered serial.

Let's call our serial (for mathematical notation's sake) s for future.

After that we have few checks and some initialization of memory. s can't be longer than 680 bits. New variable is created. Memset on some memory setting it to zero. s can't be equal to 0.

3b. **Big to array conversion**

```
.text:00401100                 lea     ebx, [ebp+var_CC]
.text:00401106
.text:00401106 loc_401106:                             ; CODE XREF: FirstCheck+D6↓j
.text:00401106                 push    edi
.text:00401107                 push    esi
.text:00401108                 lea     eax, [ebp+var_20]
.text:0040110B                 push    eax
.text:0040110C                 push    esi
.text:0040110D                 call    big_divide_qr_int
.text:00401112                 push    0
.text:00401114                 mov     [ebx], ax
.text:00401117                 push    esi
.text:00401118                 add     ebx, 2
.text:0040111B                 call    big_cmp_int
.text:00401120                 add     esp, 18h
.text:00401123                 test    eax, eax
.text:00401125                 jnz     short loc_401106
```

As we can see, this code takes s, divides it by edi (101h) and stores the reminder of

that division inside the array. This is essentially a way to convert big number to a base257 representation. The entire process repeats itself until s is equal 0.

After that fragment, s is an array which elements are all (mod 257).

3c. **Matrix multiplication**

Next part is quite essential to entire crackme, we have a simple call followed by big_set_int. Let's see what's inside the call:

```
.text:00401007                  mov     ebx, [ebp+arg_0]
.text:0040100A                  push    esi
.text:0040100B                  mov     esi, [ebp+arg_4]
.text:0040100E                  push    edi
.text:0040100F                  xor     edi, edi
.text:00401011
.text:00401011 loc_401011:                              ; CODE XREF: sub_401000+39↓j
.text:00401011                  xor     ecx, ecx
.text:00401013                  xor     eax, eax
.text:00401015
.text:00401015 loc_401015:                              ; CODE XREF: sub_401000+25↓j
.text:00401015                  movzx   edx, word ptr [ebx+ecx*2]
.text:00401019                  imul    edx, [esi]
.text:0040101C                  add     eax, edx
.text:0040101E                  inc     ecx
.text:0040101F                  add     esi, 4
.text:00401022                  cmp     ecx, 32h
.text:00401025                  jl      short loc_401015
.text:00401027                  xor     edx, edx
.text:00401029                  mov     ecx, 101h
.text:0040102E                  div     ecx
.text:00401030                  inc     edi
.text:00401031                  mov     [ebp+edi*2+var_66], dx
.text:00401036                  cmp     edi, 32h
.text:00401039                  jl      short loc_401011
.text:0040103B                  push    64h             ; size_t
.text:0040103D                  lea     eax, [ebp+var_64]
.text:00401040                  push    eax             ; void *
.text:00401041                  push    ebx             ; void *
.text:00401042                  call    _memmove
```

We see that constant array of numbers is supplied alongside our malformed serial. Then it takes i-th element of our serial multiply it by j-th element of constant array and adds it to the sum. This entire operation is repeated 32h(50) times (with constant j and i=0..50). After that the sum is divided by 257 and reminder stored in n-th cell of result array. Uff, that sounds complicated.... but if you look closecly, what it does in first loop can be translated into:

$$(s_1 \cdot c_{11} + s_2 \cdot c_{12} + s_3 \cdot c_{13} + \ldots + s_n \cdot c_{1n}) \bmod n$$

and if so, the entire call can be translated into:

$$\begin{aligned}
&(s_1 \cdot c_{11} + s_2 \cdot c_{12} + s_3 \cdot c_{13} + \ldots + s_n \cdot c_{1n}) \bmod n \\
&(s_1 \cdot c_{21} + s_2 \cdot c_{22} + s_3 \cdot c_{23} + \ldots + s_n \cdot c_{2n}) \bmod n \\
&(s_1 \cdot c_{31} + s_2 \cdot c_{32} + s_3 \cdot c_{33} + \ldots + s_n \cdot c_{3n}) \bmod n \\
&\vdots \\
&(s_1 \cdot c_{n1} + s_2 \cdot c_{n2} + s_3 \cdot c_{n3} + \ldots + s_n \cdot c_{nn}) \bmod n
\end{aligned}$$

As we can see, this is just simple multiplication of a vector by a matrix, henceforth we will use this notation to represent it:

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ \vdots \\ s_n \end{bmatrix} \times \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_n \end{bmatrix}$$

This is highschool math basically. But in case you forgot, or don't know, or don't care or god knows what else: first array is a vector $\vec{s}$, second array is a matrix A, as a result of multiplication we have $\vec{s} \times A = \vec{r}$.

As you should know, not every matrix multiplication can be reversed. First of all, for a matrix to have an inverse it needs to be square (our is). The determinant of that matrix must be different from 0 (otherwise it's a singular matrix), and most of all, there must exists such matrix B that $A \times B = I$ where I is an identity matrix (ones on the main diagonal and zeros elsewhere).

There are many ways to find a reverse matrix, but all of them are inefficient to calculate by hand on a paper when it comes to big matrices likes the one we have (50x50). So, in order to compute reverse matrix, we will use some mathematical software. I wrote a simple script for Sage that computes inverse matrix and checks if after multiplication the result is an identity matrix. The file is called matrix_inv.sage and to run it, download sage, copy the file to sage's home dir and within sage type 'load matrix_inv.sage'. As a result, you should have a file with reversed matrix. After that just some play with find and replace in some text editor and you can use it within your keygen source.

Now, that's not the end, notice that matrix multiplication takes only first 50 elements of the array and after the multiplication is done they are yet again replaced by memmove. So we have to remember that our array might be longer, but only first 50 elements will be multiplied. Ok, let's crack on.

3d. **Array to big**

```
.text:0040114E                  push    84
.text:00401150                  pop     ebx
.text:00401151
.text:00401151 loc_401151:                              ; CODE XREF: FirstCheck+11E↓j
.text:00401151                  push    edi
.text:00401152                  push    esi
.text:00401153                  push    esi
.text:00401154                  call    big_mul_int
.text:00401159                  movzx   eax, [ebp+ebx*2+var_CC]
.text:00401161                  push    eax
.text:00401162                  push    esi
.text:00401163                  push    esi
.text:00401164                  call    big_add_int
.text:00401169                  add     esp, 18h
.text:0040116C                  dec     ebx
.text:0040116D                  jns     short loc_401151
```

Well, this is as simple as it gets. Multiply big by 257, add i-th element of an array (from the end) and repeat that 84 times. This tells us that our matrix must be max 84 elements long.

Those 4 basic operations used in this keygenme are going to be useful in our keymaker as well, since from what you can see now, hopefully, reverse order uses exactly the same code (well, maybe except base36 convertion, that's why i made a python script

that will do that for us for some testing purposes, see attached base36_conv.py.

3e. The Real Algo

Here is simplified form of the main algo of this part, again pseudocode:

input:

s,n – our malformed serial and modulus

output:

c,d,a

temporary:

t,x

```
big_divide_q_2exp(t,s,400);
big_init(x);
big_set_init(x,1);
big_shift_left_int(x,x,400);
big_sub_int(x,x,1);
big_and(s,s,x);
big_destroy(x);
if big_cmp_int(t,0) then
      bad_boy;
big_mod(c,n,t);
if !big_cmp_int(c,0) then
      bad_boy
big_div(c,n,t);
```

Ok, let's explain what's happening here.

- firstly s is divided by 2^400 and the quotient of that division is stored in t (which is the equivalent of shifting that number right by 400 bits)
- then number (2^400-1) is computed
- s is and'ed with that number
- if t is 0, then go to bad boy (our input number must be greater than 2^400)
- reminder of the division of n by t is stored in c
- if c is NOT equal 0, then we go to bad bot (t must be a divisor of n)
- finally the quotient of previous division is stored in c

What does that tells us? Well, the serial must consists of 2 parts. The lower part and upper part. The upper part must be a divisor of modulus. The lower part is what's taken for earlier mentioned squaring routine. This leaves us with one important thing: we need to factorise our modulus, but hell, it's 400 bits … Like Dcoder said, 5 years ago 400bits was essentially uncrackable and now … they're putting it in crackmes … And using home machine, you can crack it quite fast i might add. I used, similarily to Dcoder, GGNFS with MSIEVE (link at the bottom). It worked for total of 15 hours on Core2Quadro 2.33 GHz, 4GB DDR3 1033MHz RAM and X25-M Intel SSD as a hard drive. I must say that I was rather surprised that it took such a short time. But it worked!

Here are the values:

```
n=B2D0FD991BDDC9E137CE7E21F04B231A3899445CC8FA6C6364BBABBED816AE77AAFFE2FC6329F64B9A
47EDAACEFF05D6EDA9
p=B37AD654A32D2D93F7D8FB81C999CEBB621B6C234666F3196B
q=FF0DBD838D1D7F92A972DA9857F35C71ABFD20F4DC00BAB63B
```

We now have a choice: we can pick p,q or number 1 as our higher part of the serial, but knowing that dealing with non cracked in half modulus later on might be problematic (there must be a reason why this was done such way, right?), we should pick one or the other. I'll go with **q**.

This is basically what happens to our serial, except the fact that after this call, we have one more thing. Let's carry on.

4. **Name manipulation.**

Let's skip for a second our powermod function, and go straight to the second check. During the name manipulation procedure, our entered name is hashed and then imported into a big number. The hash used is Whirlpool (recognised by both FindCrypt and PEiD-kanal plugins. The hash is 64 bytes long (0x40). After importing it into a big number, it is then divided by modulus and the reminder of that division is made a new big number to process later on. After that we have this code:

```
.text:0001127A                  loc_1127A:                              ; CODE XREF: SecondCheck+93↓j
.text:0001127A FF 75 0C                     push    [ebp+arg_4]
.text:0001127D 56                           push    esi
.text:0001127E E8 BD 2D 00 00               call    big_jacobi
.text:00011283 59                           pop     ecx
.text:00011284 59                           pop     ecx
.text:00011285 83 F8 01                     cmp     eax, 1
.text:00011288 75 10                        jnz     short loc_1129A
.text:0001128A FF 75 10                     push    [ebp+arg_8]
.text:0001128D 56                           push    esi
.text:0001128E E8 AD 2D 00 00               call    big_jacobi
.text:00011293 59                           pop     ecx
.text:00011294 59                           pop     ecx
.text:00011295 83 F8 01                     cmp     eax, 1
.text:00011298 74 0E                        jz      short loc_112A8
.text:0001129A
.text:0001129A                  loc_1129A:                              ; CODE XREF: SecondCheck+75↑j
.text:0001129A 6A 01                        push    1
.text:0001129C 56                           push    esi
.text:0001129D 56                           push    esi
.text:0001129E E8 9D 2B 00 00               call    big_add_int
.text:000112A3 83 C4 0C                     add     esp, 0Ch
.text:000112A6 EB D2                        jmp     short loc_1127A
```

So what does this code do? To answer this question, we have to answer the question what is Jacobi symbol. Following wikipedia:

The **Jacobi symbol** is a generalization of the Legendre symbol.
For any integer *a* and any positive odd integer *n* the Jacobi symbol is defined as the product of the Legendre symbols corresponding to the prime factors of *n*:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \left(\frac{a}{p_2}\right)^{\alpha_2} \cdots \left(\frac{a}{p_k}\right)^{\alpha_k} \quad \text{where } n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k}$$

$\left(\frac{a}{p}\right)$ represents the Legendre symbol, defined for all integers *a* and all odd primes *p* by

$$\left(\frac{a}{p}\right) = \begin{cases} 0 \text{ if } a \equiv 0 \pmod{p} \\ +1 \text{ if } a \not\equiv 0 \pmod{p} \text{ and for some integer } x,\ a \equiv x^2 \pmod{p} \\ -1 \text{ if there is no such } x. \end{cases}$$

While Legendre symbol is defined as:

Let *p* be an odd prime number. An integer a is a quadratic residue modulo p if it is congruent to a perfect square modulo p and is a quadratic nonresidue modulop otherwise. The **Legendre symbol** is a function of *a* and *p* defined as follows:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 \text{ if } a \text{ is a quadratic residue modulo } p \text{ and } a \not\equiv 0 \pmod{p} \\ -1 \text{ if } a \text{ is a quadratic non-residue modulo } p \\ 0 \text{ if } a \equiv 0 \pmod{p}. \end{cases}$$

Knowing that, we deduce that parameters to big_jacobi should be our m (big representation of hashed name) and a prime number p. When we look at our parameters, we notice that number p depends on what we typed in as a serial (and more specifically the higher part of our serial). Had we chosen our higher part to be 1, the other number would be exactly n, and that is not a prime number. Most big num libraries say that with non prime numbers the result of legendre/jacobi symbol is undefined. So this is the reason, why higher part of our serial must be one of the factors of modulus. Thanks to that we can compute Jacobi symbol for our name over one of the factors. This combined with definition of Jacobi symbol from previous page, tells us that if Legendre symbol for every prime factor (different than 1) is equal to 1, then the Jacobi symbol of a number over n will also be 1. But what does it give us? It tells us that this specific number is a quadratic residue modulo n. In other words, there exists a number x such that $a \equiv x^2 (mod\ n)$. This is the most vital part for us.

If you remember from the start, our modified serial is squared modulo n. Now, with the knowledge of name modification and small fragment of code that shortly follows:

```
.text:0001132F 8D 45 DC          lea     eax, [ebp+var_24]
.text:00011332 50                push    eax
.text:00011333 8D 45 F4          lea     eax, [ebp+var_C]
.text:00011336 50                push    eax
.text:00011337 E8 94 34 00 00    call    big_compare
.text:0001133C 83 C4 2C          add     esp, 2Ch
.text:0001133F 8B F0             mov     esi, eax
.text:00011341 F7 DE             neg     esi
.text:00011343 1B F6             sbb     esi, esi
.text:00011345 46                inc     esi
```

we can conclude that our name must be equal to the square of our serial modulo n, and the Jacobi symbol on each factor of n provides us with a number that is for sure a square power of some number x. Thus we finally know what is our main goal in this crackme. We need to find such x, that raised to the second power modulo n will give us the same number as our hashed name.

5. **Square root computation**

Solving this problem is basically reduced to a convention wether the modulus is a prime number, power of a prime or composite modulus. In Handbook of Applied Cryptography the algorithm used (mainly for the needs of Rabin cryptosystem) is slightly modified for a specific example. It deals with a situation when modulus is a composite number that is a result of multiplication two big prime numbers (n=pq). That's exactly the problem we have to solve. Allow me to quote from this great book:

3.44 Algorithm Finding square roots modulo n given its prime factors p and q

INPUT: an integer n, its prime factors p and q, and $a \in Q_n$.
OUTPUT: the four square roots of a modulo n.

1. Use Algorithm 3.39 (or Algorithm 3.36 or 3.37, if applicable) to find the two square roots r and $-r$ of a modulo p.
2. Use Algorithm 3.39 (or Algorithm 3.36 or 3.37, if applicable) to find the two square roots s and $-s$ of a modulo q.
3. Use the extended Euclidean algorithm (Algorithm 2.107) to find integers c and d such that $cp + dq = 1$.
4. Set $x \leftarrow (rdq + scp) \bmod n$ and $y \leftarrow (rdq - scp) \bmod n$.
5. Return$(\pm x \bmod n, \pm y \bmod n)$.

Ok, but it says we need to find square roots of that number modulo p and q respectively. To do that we again have couple of algorithms. They all depend how our prime is build. I'll make it slightly faster and tell you that there are two special cases to compute it when prime p is congruent to 3 (mod 4) or 5 (mod 8). In our case it's 3 (mod 4). And the algorithm for this, is as follow:

3.36 Algorithm Finding square roots modulo a prime p where $p \equiv 3 \pmod 4$

INPUT: an odd prime p where $p \equiv 3 \pmod 4$, and a square $a \in Q_p$.
OUTPUT: the two square roots of a modulo p.

1. Compute $r = a^{(p+1)/4} \bmod p$ (Algorithm 2.143).
2. Return$(r, -r)$.

There is also one thing. After we solve this, we have this pair of equations:
$r^2 \bmod p = a$
$s^2 \bmod q = a$
We can easily notice that we can apply Chinese Reminder Theorem to find such **x** that will meet the required eqation:
$x^2 (mod\ pq) = a$
And the result of that CRT will be our needed **x**!

6. **Coding a keymaker**

Well, finally we are able to create a working keygen. To do so, we need to make steps:
Input:

n - modulus
p - prime factor of n
q - prime factor of n
M - name number

Output:

S - serial

Proceedings:

- Calculate Whirlpool hash of name
- Import it to a big number M
- Divide it by n and replace M with the reminder
- Add one to M until $\left(\frac{M}{p}\right)=1$ and $\left(\frac{M}{q}\right)=1$

- Solve the square root for M and replace it
- Initialise S with p or q (choice is yours) and shift it left by 400 bits
- Add M to S
- Convert S to a vector (mentioned earlier)
- Multiply that vector by inverted matrix A^{-1}
- Copy the result back to the vector
- Convert the vector to a big number S
- Convert S to a Base36 string representation

After you've done it all, you will have a fully working keygen! There's not much magic to it, code is quite simple as you can see it inside this solution. Also inside you have two small scripts that i found useful, first one is a python script that converts any given number to a base36 string and the other one is a SAGE script to compute inverted matrix needed to compute the correct serial. I also include patched version of the crackme with removed annoyances.

Cheers go out to all people who are keen to gain knowledge!
And also to all folks on #crackmesde @ DALnet
And of course to waganono for creating such an awesome keygenme!

If you have any questions regarding this tutorial or RCE at all, feel free to poke me on my webpage (tamaroth.eu) or on crackmes.de