[kEYGENING RSA-512]
OR
[MAKING A KEYGEN FOR bart's PRGN cRACKME]

(c) by [bLaCk-eye/K23]

INTRO:

It's been quite a while since i managed to solve this crypto crackme, but didn't strugle to much to make a tutorial because it was already solved and becuase i didn't have the time required to write a tutorial and in the end becuase other target's caught my attention.But my 'luck' is that i almost had a fracture at my right foot so i'm home 10 days enjoying the free time.This brings me allot of memories: i began cracking after fracturing my left arm :), but at that time i was going to school.Hope you and enjoy this tutorial.

One of the final reasons for releasing this tutorial is showing you a new great method of finding the calls in the miracl bignum library, a method is very fast and almost fail proof when it can be apllied.The method was brought to my attention by bRain_faKKer so we owe him the discovery of this method,which i'll explain through out this tutorial.

TOOLS:

- IDA: for the best dissasembly possible of the crackme;
- Miracl Library (optional);

DIFICULTY:

I would say 2/10 but it mostly depends on the readers/crackers crypto knowledge.

TUTORIAL:

Load the program in any file identifier and we see it's a made with LCC,not packed or anything.Load it up in ida and wait for the completion of the dissasembly process.

After that's done,look like always through the strings references and will find one usefull hint:

aMiraclNotIniti db 'MIRACL not initialised - no call to mirsys()

Now that we know the program uses miracl library we need to discover what each function of the protection is (e.g recognise the miracl function).At first i tried to use my miracl.sig file from the bignum signature package,but with no great succes:a few function were recognized but none of the protection it's self.So i'll give now the promised method.

If you ever dissasembled or if you are familiar with the miracl package you almost sure know that most of the functions can be recognised by a 'magic number' which apears in the body of the function.The particular thing is that the number is unique for that function and doesn't change thought the different version of the library.Wanna see?

- for mirvar:
  from source: **MR_IN(23);**
  from ida: **mov dword ptr [esi+eax*4+20h], 23;**
- for set_io_buffer_size:
  from source: **MR_IN(142);**
  from ida: **mov dword ptr [esi+eax*4+20h], 142;**

To have an easy job in the future just do like i did: create a file with name+magic key of every function.

So when you have to keygen for a crypto-keygenme, first thing you assure yourself that it uses miracl and then when in need for identifying a function just look in the function for this instruction:

```
mov     dword ptr [esi+eax*4+20h], XX
```

Take XX value and search it in your magic table and you have yourself the function. It's easy isn't it?One minor thing is that some of the function don't have a magic number but those are,as i observed, functions that are from the raw core of miracl,this means function to allocate memory,setup things and other stuff.

Let's get to the protection:

```
.text:004013A1                  push    100h            ; nMaxCount
.text:004013A6                  push    offset unk_40C148 ; lpString
.text:004013AB                  push    65h             ; nIDDlgItem
.text:004013AD                  push    edi             ; hDlg
.text:004013AE                  call    GetDlgItemTextA
.text:004013B3                  mov     ds:dword_40B92C, eax
.text:004013B8                  push    100h            ; nMaxCount
.text:004013BD                  push    offset byte_40B018 ; lpString
.text:004013C2                  push    67h             ; nIDDlgItem
.text:004013C4                  push    edi             ; hDlg
.text:004013C5                  call    GetDlgItemTextA
```

This get's the name and serial.

Let's go on:

```
.text:004013FB                  mov     eax, ds:dword_40B008
.text:00401400                  movzx   eax, ds:byte_40B018[eax]
.text:00401408                  mov     ds:dword_40B00C, eax
.text:0040140D                  cmp     eax, 41h
.text:00401410                  jb      short loc_401417
.text:00401412                  cmp     eax, 46h
.text:00401415                  jbe     short loc_401438
.text:00401417
.text:00401417 loc_401417:
.text:00401417                  mov     eax, ds:dword_40B00C
.text:0040141C                  cmp     eax, 30h
.text:0040141F                  jb      short loc_401426
.text:00401421                  cmp     eax, 39h
.text:00401424                  jbe     short loc_401438
.text:00401426
.text:00401426 loc_401426:
.text:00401426                  push    offset aInvalidKey ; lpString
.text:0040142B                  push    67h             ; nIDDlgItem
.text:0040142D                  push    edi             ; hDlg
.text:0040142E                  call    SetDlgItemTextA
.text:00401433                  jmp     loc_401593
```

This code checks that the serial is in hex format (0123...DEF)

Let's go on (the miracl function are already indentified by me using the method explained previos and some local variable are renamed after their use):

```
.text:00401452                  push    0
.text:00401454                  push    0FA0h
.text:00401459                  call    mirsys
.text:0040145E                  mov     [ebp+var_4], eax
.text:00401461                  push    0
.text:00401463                  call    mirvar
.text:00401468                  mov     ds:big_result, eax
```

```
.text:0040146D                  push    0
.text:0040146F                  call    mirvar
.text:00401474                  mov     ds:big_modulus, eax
.text:00401479                  push    0
.text:0040147B                  call    mirvar
.text:00401480                  mov     ds:big_e, eax
.text:00401485                  push    0
.text:00401487                  call    mirvar
.text:0040148C                  mov     ds:big_serial, eax
.text:00401491                  push    0
.text:00401493                  call    mirvar
.text:00401498                  mov     ds:big_name, eax
```

This code initilises miracl and creates 5 bignums.
Next:

```
.text:004014AA                  push    ds:big_name
.text:004014B0                  push    offset name
.text:004014B5                  push    ds:len_name
.text:004014BB                  call    bytes_to_big
.text:004014C0                  push    offset serial
.text:004014C5                  push    ds:big_serial
.text:004014CB                  call    cinstr
.text:004014D0                  push    offset modulus
.text:004014D5                  push    ds:big_modulus
.text:004014DB                  call    cinstr
.text:004014E0                  push    ds:big_e
.text:004014E6                  push    10001h
.text:004014EB                  call    _convert
.text:004014F0                  push    ds:big_result
.text:004014F6                  push    ds:big_modulus
.text:004014FC                  push    ds:big_e
.text:00401502                  push    ds:big_serial
.text:00401508                  call    powmod
.text:0040150D                  push    ds:big_result
.text:00401513                  push    ds:big_name
.text:00401519                  call    _compare
.text:0040151E                  add     esp, 58h
.text:00401521                  or      eax, eax
.text:00401523                  jnz     short loc_40152F
.text:00401525                  mov     ds:flag, 1
```

So now it put's the name as a bignum,reads the serial and the moduluis as a hex bignum.Will find later what's the value of modulus.

Then it put's a prime integer (10001h) in a bignum.If you are familiar with rsa you would know that this number is the most used number as the encryption exponent.After this it performs a powmod operation:

big_result = big_serial^10001h mod big_modulus

It certainly looks like rsa :)

If you execute the crackme in a debugger you'll see that the modulus it's our magic key generated by the program;so let's see how it's generated.For this we need to go into the WM_INITDIALOG function:

```
.text:004015BB                  call    GetTickCount
.text:004015C0                  push    eax
.text:004015C1                  push    [ebp+hDlg]
.text:004015C4                  call    sub_4012D5
```

So it get's the number of miliseconds since windows has been started and then calls a procedure;let's go in it:

```
.text:004012D5                  push    ebp
.text:004012D6                  mov     ebp, esp
.text:004012D8                  push    edi
.text:004012D9                  push    0
.text:004012DB                  push    0FA0h
.text:004012E0                  call    mirsys
.text:004012E5                  mov     edi, eax
.text:004012E7                  mov     dword ptr [edi+238h], 10h
.text:004012F1                  push    0
.text:004012F3                  call    mirvar
.text:004012F8                  mov     ds:dword_40B11C, eax
.text:004012FD                  push    0
.text:004012FF                  call    mirvar
.text:00401304                  mov     ds:dword_40C13C, eax
.text:00401309                  push    0
.text:0040130B                  call    mirvar
.text:00401310                  mov     ds:big_modulus, eax
.text:00401315                  push    [ebp+rnd]
.text:00401318                  call    ChangeSalt
```

So we see again the initialisation of miracl and the creation of 3 bignums.

Then we see a call to procedure that makes from the dword random salt (GetTickCount) a word rnd value

Then:

```
.text:0040131D                  push    ds:dword_40B11C
.text:00401323                  push    20h
.text:00401325                  call    sub_401297
.text:0040132A                  push    ds:dword_40C13C
.text:00401330                  push    20h
.text:00401332                  call    sub_401297
.text:00401337                  push    ds:big_modulus
.text:0040133D                  push    ds:dword_40C13C
.text:00401343                  push    ds:dword_40B11C
.text:00401349                  call    multiply
.text:0040134E                  push    offset modulus
.text:00401353                  push    ds:big_modulus
.text:00401359                  call    big_to_bytes
.text:0040135E                  push    offset modulus  ; lpString
.text:00401363                  push    66h             ; nIDDlgItem
.text:00401365                  push    [ebp+hDlg]      ; hDlg
.text:00401368                  call    SetDlgItemTextA
```

Inside sub_401297:

```
.text:00401297                  push    ebp
.text:00401298                  mov     ebp, esp
.text:0040129A                  push    ebx
.text:0040129B                  push    esi
.text:0040129C                  push    edi
.text:0040129D                  mov     esi, [ebp+arg_0]
.text:004012A0                  mov     ebx, [ebp+arg_4]
.text:004012A3                  xor     edi, edi
.text:004012A5                  jmp     short loc_4012B6
.text:004012A7 fill_prime_array:
```

```
.text:004012A7                  call    sub_40126A
.text:004012AC                  mov     edx, eax
.text:004012AE                  mov     ds:prime_array[edi], dl
.text:004012B5                  inc     edi
.text:004012B6
.text:004012B6 loc_4012B6:
.text:004012B6                  cmp     edi, esi
.text:004012B8                  jb      short fill_prime_array
.text:004012BA                  push    ebx
.text:004012BB                  push    offset prime_array
.text:004012C0                  push    esi
.text:004012C1                  call    bytes_to_big
.text:004012C6                  push    ebx
.text:004012C7                  push    ebx
.text:004012C8                  call    _nxprime
.text:004012CD                  add     esp, 14h
.text:004012D0                  pop     edi
.text:004012D1                  pop     esi
.text:004012D2                  pop     ebx
.text:004012D3                  pop     ebp
.text:004012D4                  retn
```

So now it's obvious that this routine creates a random 256bit table from the 16bit random value;then it put's it in a bignum and increments it untill a prime number is reached.

So the modulus is the product of two random prime numbers.

Now we can say for sure that the protection of the crackme consists in a runtime RSA-512.

For us to make a keygen we need to find out the decryption exponent: D.

If we would only have the modulus and the encryption exponent like in the regular rsa it would have been impossible for us to make a keygen coz it we would have needed to factor the 512 bit modulus.

But we have an advantage:we know that each of the primes that make up the modulus are generated from a random 16bit value and how the prime numbers are built starting from this random number.

So to make a keygen:

1. We take user name to generate serial
2. We take the magic key from the crackme
3. We take each value from 0 to FFFFh and create a prime bignumber to see if it's a factor of the magic key.We have 2^16 posibilities so it should be pretty fast.After step we should have the two factors of modulus: q and p
4. We now need to find the decryption exponent.This is done by solving:
   d*e=1 mod [(p-1)(q-1)]
   d= e^(-1) mod [(p-1)(q-1)]
5. We have
   serial=name^d mod modulus

This is all we need to do.

Nice crackme from bart.

Regards,

bLaCk-eye

PS:Check the source of the keygen if you failed to understand something.

GREETZ:

- all Kanal23 members (www.kanal23.knows.it)
- all tkm members

- everybody that knows me and bares with me and my stupid questions sometimes: you know who you are.

mycherynos@yahoo.com