

**[KEYGENING THE dEAMON]
OR
[HOW TO MAKE A KEYGEN FOR TKM! TRIAL 2 by _ged]**

by [bLaCk-eye]/K23

INTRO:

In this tutorial we are going to handle a target which at a first look seems very tough to defeat (it seemed so to me, bLaCk). We are writing this because the trial expired few days ago and we think that releasing it won't harm anything or anyone and because we know that tkml has already a new kick-ass trial crackme, trust us we are very good informed. Let's get the monster screaming like a little girl :).

TOOLS:

- IDA for dissassembling and getting a good source code
- a tool to unpack the fsg compressed trial: we use imprec but you can use any fsg generic or not unpacker
- PEiD for indentifying the packer used but if you have any experience with packers you will surely recognize fsg 1.33

DIFFICULTY:

Let's say 4-5/10

TUTORIAL:

So after having a first look at the crackme we see it's a very common name - serial protection scheme.

It's packed with Fsg.133 so get your unpacker or use like me Imprec to get a working, unpacked executable. We won't go into details on how to use imprec, just read the help file, this is a tutorial on the protection scheme not about unpacking a packed target.

Done? Examining the unpacked target we see that it's written in LCC or that is what PEiD tells us. Ok load the file into IDA, wait a couple of minutes for it to completely finish the dissassembly process and now look into the dissassembly for any hints on what the trial is doing or using. Already we find some interesting strings:

```
a10c3e43f01a319    db
'10C3E43F01A3197EFBAA23C4910D954F46FD68D904FEAD0851BB3AB79811'
'D02F19979412EFF3A116787E3C9C646A35291E6A739F975E57A7DBD40A59'
'1138AC4A',0
aC5ee2c226dbc49    db
'C5EE2C226DBC49B818F9E4A1C6C20A4D50F6E2DE78A30FC0CF277304EC86'
'3BDE71C78BCF48E4826179751F68E7B114F2C2107503CE955E596828BA06'
'0C07C9FC',0
a1ec4def070ae4e    db
'1EC4DEF070AE4E7178996A7218D5DACB876F03307214C70E1085CF36ADC1'
'F34592F826BAA462E7A50921134E01467841D47BB8A42E9E6FE934B3BE25'
'204D8F85',0
a3094effb1e0b05    db
'3094EFFB1E0B05C8978D0F81EB89366215F5881803D3BD1C90400E418E10'
'8857BAFB71FD034340A346B5F36CBB009F745FE3308498C4484D00E732F5'
'AA33D5F1',0
aC93fb42656ec63    db
'C93FB42656EC633ED4428363F9E3146D6D9A94365113775898884A3886B5'
'A3E9CC54E913E1DE8642FF80D35179AA39918B3902FA209C0FB417240747'
```

```
'01ADDCD3',0
aD7eae4a5b34196 db
'D7EAE4A5B34196FFF0B433297640FDD7438891BBFB284B1F7434F79CBECC'
'79CD6C100E2652A3A83D934D27F18B242F4D344EA5F049940094DF0DD53B'
'AECEFA5B',0
```

Now that are some really big bignums,512 bit more precisely, i can tell you that some are primes some not but that doesn't really present any interest for us at the present moment. The presence of the bignums assures us of the presence of a bignum library of course. Now we need to identify the bignum, and this is easy too look down a little:

```
aMiraclNotIniti db 'MIRACL not initialised - no call to mirsys
()',0Ah,0
```

Ok so now we now that the trial is using the well known Miracl package and that represents an advantage on our side because if _ged had used an unknown biglib we would have big trouble trying to indentify correctly the procedures.

Know having in mind that we have to deal with recognizing the procedures.We have two options:

- ?? recognize them ourself, you can do that if you are aquinted with miracl and you know the procedures by heart (not trully our case)
- ?? i remebered that on tkm! site there is a miracl lib that the members of tkm put there for use to compile the many sources from their site, so i thought not to download that miracle lib, using flair to create a signature file and apply it to our database and see the results.

I went with of course path number two because it seemed more convient and less time consuming.It's easy to make a signature file if you have the flair package that you can download from ida's site if you are a proud owner of a registered version, not like me.Again i won't explain what you need to do for the already specified reason, but if you somehow couldn't manage it,you'll find allong with this tutorial and the sources of the keygen the signature file so you can use it yourself.Now that being said apply the sig file and be proud coz the flair recognized 81 functions, more then enough.

Let's find the protection scheme:

```
loc_4027C5: ; CODE XREF: DialogFunc+50 j
    push 63h ; nMaxCount
    push offset String ; lpString
    push 0C8h ; nIDDLgItem
    push [ebp+hWnd] ; hDlg
    call GetDlgItemTextA
    mov ds:dword_40B004, eax
    cmp ds:dword_40B004, 0
    jnz short loc_402800
    push 0 ; uType
    push offset aPfff__ ; lpCaption
    push offset aEnterName ; lpText
    push [ebp+hWnd] ; hWnd
    call MessageBoxA
    jmp loc_402902

loc_402800: ; CODE XREF: DialogFunc+83 j
    push 200h ; nMaxCount
    push offset dword_40C178 ; lpString
    push 0C9h ; nIDDLgItem
    push [ebp+hWnd] ; hDlg
    call GetDlgItemTextA
    mov ds:dword_40B004, eax
    cmp ds:dword_40B004, 0
```

```

    jnz     short loc_40283E
    push    0                ; uType
    push    offset aPfff__   ; lpCaption
    push    offset aEnterSerial ; lpText
    push    [ebp+hWnd]      ; hWnd
    call    MessageBoxA
    jmp     loc_402902

loc_40283E:
                ; CODE XREF: DialogFunc+C1 j
    push    offset dword_40C178
    push    offset String
    call    sub_4022C5
    add     esp, 8
    cmp     eax, 0
    jnz     short loc_40286E
    push    0                ; uType
    push    offset aJoeSays   ; lpCaption
    push    offset aCongratulation ; lpText
    push    [ebp+hWnd]      ; hWnd
    call    MessageBoxA

```

So the checking call is this one: call sub_4022C5, and we see that after it there is the check good guy compare. Let's go into the call and analyse it into pieces to understand. I warn that some of the local variables have been renamed by me to make understanding easy:

```

    lea     edi, [ebp+_name_md5]
    lea     esi, ds:40C378h
    mov     ecx, 10h
    rep movsb
    lea     edi, [ebp+_hex_serial]
    lea     esi, ds:40C388h
    mov     ecx, 80h
    rep movsb
    lea     edi, [ebp+var_90]
    lea     esi, ds:40C408h
    mov     ecx, 80h
    rep movsb
    push    0
    push    1F4h
    call    _mirsys
    add     esp, 8
    mov     [ebp+var_19C], eax
    mov     edi, [ebp+var_19C]
    mov     dword ptr [edi+238h], 10h
    push    80h
    lea     edi, [ebp+_hex_serial]
    push    edi
    call    RtlZeroMemory
    push    [ebp+_serial]
    lea     edi, [ebp+_hex_serial]
    push    edi
    call    ascii2hex
    add     esp, 8
    lea     edi, [ebp+MD_ctx]
    push    edi
    call    MD5Init
    add     esp, 4
    push    [ebp+_name]

```

```

call    strlen
add     esp, 4
push    eax
push    [ebp+_name]
lea     edi, [ebp+MD_ctx]
push    edi
call    MD5Compress
add     esp, 0Ch
lea     edi, [ebp+MD_ctx]
push    edi
lea     edi, [ebp+_name_md5]
push    edi
call    MD5Finish
add     esp, 8

```

So this code initializes the miracl library, converts the serial into hex and hashes the name with a modified md5 hash, more precisely only the init values are changed into the name of the group: "TERAZ KURWA MY!!".

Let's go:

```

push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_1], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_2], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_3], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_4], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_5], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_6], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_7], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_8], eax
push    0
call    _mirvar
add     esp, 4
mov     [ebp+big_9], eax
push    0
call    _mirvar
add     esp, 4

```

```

mov    [ebp+big_10], eax
push   0
call   _mirvar
add    esp, 4
mov    [ebp+big_11], eax
push   0
call   _mirvar
add    esp, 4
mov    [ebp+big_12], eax

```

So it initialises 12 bignums for later or sooner use as we see next:

```

push   offset aD7eae4a5b34196 ; bignum
push   [ebp+big_1]
call   _cinstr
add    esp, 8
push   offset aC93fb42656ec63 ; bignum
push   [ebp+big_2]
call   _cinstr
add    esp, 8
push   offset a3094effb1e0b05 ; bignum
push   [ebp+big_5]
call   _cinstr
add    esp, 8
push   offset a1ec4def070ae4e ; bignum
push   [ebp+big_6]
call   _cinstr
add    esp, 8
push   offset aC5ee2c226dbc49 ; bignum
push   [ebp+big_8]
call   _cinstr
add    esp, 8
push   offset a10c3e43f01a319 ; bignum
push   [ebp+big_7]
call   _cinstr
add    esp, 8
push   [ebp+big_9]
lea    edi, [ebp+_hex_serial]
push   edi
push   80h
call   _bytes_to_big

```

So it puts the 6 bignum strings that we saw at the beginning into 6 bignums and the serial in hex format into another bignum.

```

push   [ebp+big_10]
push   [ebp+big_1]
push   [ebp+big_5]
push   [ebp+big_9]
call   _powmod
add    esp, 10h
push   [ebp+big_11]
push   [ebp+big_2]
push   [ebp+big_6]
push   [ebp+big_9]
call   _powmod
add    esp, 10h
push   [ebp+big_10]
push   [ebp+big_11]

```

```

call  _compare
add   esp, 8
mov   ds:flag, eax
cmp   ds:flag, 0
jge   loc_4025C7

```

So we have two powmod operations and one compare one,let's see:

```

big_10=big_9^big_5 mod big_1
big_11=big_9^big_6 mod big_2

```

,where big_9=our serial

Now is the compare operation: if big_11>=big_10 it jumps to another piece of code else it continues.So we have a branch as it is called.Let's see what happens when big_10>big_11:

```

mov   edi, [ebp+big_10]
push  edi
push  [ebp+big_11]
push  edi
call  _subtract
add   esp, 0Ch
mov   edi, [ebp+big_10]
push  edi
push  [ebp+big_8]
push  edi
call  _multiply
add   esp, 0Ch
mov   edi, [ebp+big_1]
push  edi
push  edi
push  [ebp+big_10]
call  _divide
add   esp, 0Ch
mov   edi, [ebp+big_10]
push  edi
push  [ebp+big_2]
push  edi
call  _multiply
add   esp, 0Ch
mov   edi, [ebp+big_10]
push  edi
push  [ebp+big_11]
push  edi
call  _add
add   esp, 0Ch
push  [ebp+big_12]
push  [ebp+big_10]
call  _copy
add   esp, 8
jmp   loc_402649

```

So the operations done here are:

```

big_10 = big_10 - big_11
big_10 = big_10 * big_8
big_10 = big_10 mod big_1
big_10 = big_10 * big_2
big_10 = big_10 + big_11

```

`big_12 = big_10`

And the after the jump (to a portion of code that is common to both):

```
loc_402649:                ; CODE XREF: Check+2FD j
    push    1
    lea     edi, [ebp+var_90]
    push    edi
    push    [ebp+big_12]
    push    80h
    call    _big_to_bytes
    add     esp, 10h
    mov     byte ptr [ebp+_name_md5], 1
    mov     ds:i, 0

loc_402673:                ; CODE XREF: Check+3EA j
    push    10h
    mov     edi, ds:i
    shl     edi, 4
    lea     edi, [ebp+edi+var_90]
    push    edi
    lea     edi, [ebp+_name_md5]
    push    edi
    call    memcmp
    add     esp, 0Ch
    mov     ds:flag, eax
    cmp     ds:flag, 0
    jz      short loc_4026A2
```

So then it exports the bignum as a 256 base string

It changes the first byte of the hash into 01h and check if the exported bignums is made by eight md5hashes. If it's like this then we are registered

If we follow the other code, the one that is followed if `big_11 >= big_10` we see that it's similar to the previous code that is:

```
big_11 = big_11 - big_10
big_11 = big_11 * big_7
big_11 = big_11 mod big_2
big_11 = big_11 * big_1
big_11 = big_11 + big_10
big_12 = big_11
```

So this is all the code of the registration routine:

```
big_10=big_9^big_5 mod big_1
big_11=big_9^big_6 mod big_2
```

```
if big_11 >= big_10
((big_11-big_10)*big_7) mod big_2 ) * big_1 + big_10) = 8 times md5
_hash
else
((big_10-big_11)*big_8) mod big_1 ) * big_2 + big_11) = 8 times md5
_hash
```

So what those two equation represent is a rsa decryption using not the standard powmod operation but using the Chinese Remainder Theorem(CRT). I'll explain how this is done:

Say that we have a rsa modulus **n** and **a** decryption key d, then to decrypt something we need to perform $x^d \bmod n$, where x is from $Z(n)$. Instead of doing directly that we can be

quicker doing like this:

```
*say that n=p*q
*we calculate
- dp = d mod (p-1)
- dq = d mod (q-1)
*solving Y so that both equations are true at the same time
Y = x^dp mod p
Y = x^dq mod p
```

If we use the CRT we find out that $Y = x^d \bmod (p \cdot q)$.

The major advantage over the classical powmod is that it's about 4 times faster.

In our case the values are like this:

```
p=D7EAE4A5B34196FFF0B433297640FDD7438891BBFB284B1F7434F79CBEC79CD6C1
00E2652A3A83D934D27F18B242F4D344EA5F049940094DF0DD53BAECEFA5B'
```

```
q=C93FB42656EC633ED4428363F9E3146D6D9A94365113775898884A3886B5A3E9CC5
4E913E1DE8642FF80D35179AA39918B3902FA209C0FB41724074701ADDCD3
```

```
dp=
3094EFFF1E0B05C8978D0F81EB89366215F5881803D3BD1C90400E418E108857BAFB7
1FD034340A346B5F36CBB009F745FE3308498C4484D00E732F5AA33D5F1
```

```
dq=
1EC4DEF070AE4E7178996A7218D5DACB876F03307214C70E1085CF36ADC1F34592F82
6BAA462E7A50921134E01467841D47BB8A42E9E6FE934B3BE25204D8F85
Y= 8 times the md5 hash
```

So what the crackme does is decrypt the 8 times md5 hash and then compare with our serial. We need to hash the name, copy it 8 times and encrypt it with rsa-1024 (yeah but we have the factors so don't worry).

Now we know that

```
Y = x^dp mod p
Y = x^dq mod p,
where Y=8 times md5
x=serial
```

Knowing that in rsa $e \cdot d \bmod (p-1)(q-1) = 1$, it's quite obvious to see that to find x we have to solve this:

```
x=Y^(dp^(-1) mod (p-1)) mod p
x=Y^(dq^(-1) mod (q-1)) mod q
```

So we need to calculate $ep = dp^{-1} \bmod (p-1)$ and $eq = dq^{-1} \bmod (q-1)$:

```
ep=
9FD6A8F869859893655F930FA3976E79C20295AC83585329C706364F5F3A0D662C2F9
561FA2E11AC495FEEBBECAC360988498ACC8F4FDB9CF8EF730A10AD3815
eq=
26DBADCBB714E0288EACA30D18744AA7DDF44662363C8B3E80558C9A173B0A20924D3
CC4E36426A31D2D90F075DB7F4BF1F640BEE7AE84D73DCD0686EEAB8DB9
```

Now we have everything to make a keygen:

1. Get name
2. Hash name with modified md5 and change the first byte to 01h
3. Copy the hash 8 times in a table

4. Put the table in bignum Y and solve using CRT:
 $x = Y^{(dp^{-1} \bmod (p-1)) \bmod p}$
 $x = Y^{(dq^{-1} \bmod (q-1)) \bmod q}$
5. The serial is x.

That's it. If you have any problems understanding this tutorial please look through the keygen source and then mail me if the question remains.

Regards,

[bLaCk-eye/K23](#)

Greetz:

- All Kanal23 members and especially bRain faKker for helping me
- All TKM! members and especially _ged for bringing us this interesting challenge
- All the dudes in #reversing4elites: you are trully elite reversers
- All the dudes in #compression

Contact:

- efnet: #kanal23, #reversing4elites
- mail: mycherynos@yahoo.com