Cracking Veneta’s Insipid Crackme v.2

by bLaCk-eye

www.cryptocracking.cjb.net

Intro:
Welcome to this new tutorial from me. Today’s target is of course another crypto crackme published on www.crackmes.de some days ago. As the author says, it’s directed to newbies in cryptography.
Have a nice reading…

Info:

 Crackme……………Insipid Crackme v.2
 Author………………Veneta/MBE
 Url…………………www.crackmes.de
 Type………………..crypto keygenme
 Protection…………..sha256, e2, CRT-256
 Difficulty…………...2/10

Tools:
- IDA
- OllyDbg
- MSVC 6.0 for keygen
- FSG Unpacker
- Crypto Searcher by x3chun (can get it gere: www.cryptocracking.cjb.net )

Essay:

Ok, load the target in PEiD and we all see it’s compressed using fsg 2.0. Grab an unpacker for it, or anything to obtain an unpacked version of the crackme.
As soon as we have an unpacked version, we can use the crypto searcher to check for any known crypto algorithm signature. The results are quite satisfactory:
- SHA 256
- E2 by Nippon Telegraph
- Biglib

The above information will be very useful when coding the keygen.
Now load it in IDA and wait for the disassembling and analysis to end.
Once Ida is finished you can use the information gathered by the crypto searcher. From the fact that the crackme is coded in assembly we figure that the crypto algo might be written by Witeg ( www.witeg.cad.pl ), as he is the only one who has implemented the algo’s in assembly. Instead of downloading the sources from Witeg’s site and then trying to figure out what procedure represents what we go www.cryptosig.prv.pl , a site maintained by Cauchy. The site is about a crypto signature to use with IDA (.sig) to
identify the crypto routines. What did Cauchy do actually: he has taken all of Witeg’s public sources (yes there are some unreleased ones, which I got my hands on) and compiled then and from the result of the compile he built a signature file for IDA. (for more details on signature files, what they represent and how to use them, go to Ida’s site). This signature file will recognize itself and automatically any of the Witeg’s routines used in the crackme.

To use the signature file, download it from Cauchy’s site, extract it into Ida\Sig\ directory and in ida press SHIFT+F5, then INS and from the list of available signature file select “Crypto by Cauchy//HTB Team”.

Ida will apply it and you’ll see it has found 6 functions.

Now let’s ge finally to the core of the protection (many of the variables have been renamed by myself to improve the quality of the disassembly):

- crackme get’s the name of the user and checks if it’s greater then zero but smaller then 32 in leght:

```assembly
.RIF1:0040454A          push    100h            ; nMaxCount
.RIF1:0040454F          push    offset szname   ; lpString
.RIF1:00404554          push    3ECh            ; nIDDlgItem
.RIF1:00404559          push    ds:hDlg         ; hDlg
.RIF1:0040455F          call    GetDlgItemTextA
.RIF1:00404564          test    eax, eax
.RIF1:00404566          jz      loc_404715
.RIF1:0040456C          cmp     eax, 20h
.RIF1:0040456F          ja      loc_404715
```

- the crackme hashes the name with Sha256 to get a hash of 256bits

```assembly
.RIF1:00404575          push    offset szname
.RIF1:0040457A          push    eax
.RIF1:0040457B          push    offset name_sha1 ; address of hash
.RIF1:00404580          call    _SHA256@0       ; SHA256()
```

- the crackme then encrypts the hash using E2 and the following 16 bytes key: “.:exile:.”,0x0,”A0F792”. The hash is encrypted in 128 bit blocks (that’s the size of the blocks used by E2 cipher, check it’s specifications).

```assembly
.RIF1:00404585          push    offset a_Exile_ ; ":.exile:.
.RIF1:0040458A          call    _E2_SetKey@4    ; E2_SetKey(x)
.RIF1:0040458F          push    offset name_sha1
.RIF1:00404594          push    offset e2_encrypted_name_hash_1
.RIF1:00404599          call    _E2_Encrypt@8   ; E2_Encrypt(x,x)
.RIF1:0040459E          push    offset name_sha_2
.RIF1:004045A3          push    offset e2_encrypted_name_hash_2
.RIF1:004045A8          call    _E2_Encrypt@8   ; E2_Encrypt(x,x)
.RIF1:004045AD          call    _E2_Clear@0     ; E2_Clear()
```

- the crackme gets the serial number and checks if it’s length is greater then zero:

```assembly
.RIF1:004045B2          push    100h            ; nMaxCount
.RIF1:004045B7          push    offset szserial ; lpString
.RIF1:004045BC          push    3EFh            ; nIDDlgItem
.RIF1:004045C1          push    ds:hDlg         ; hDlg
.RIF1:004045C7          call    GetDlgItemTextA
.RIF1:004045CC          test    eax, eax
.RIF1:004045CE          jz      loc_404715   ; bad jump
```

- the crackme then creates 8 bignums, used later:
- some of the bignum are getting initialized:

- now that the bignums are initialised the crackme uses them to test the serial number:
So what is exactly do in the above code?
Well it calculates:

rem_1 = big_serial % prime1
rem_2 = big_serial % prime2, "%" represents modulo operation (gives the reminder)

If:
rem_1 == sha256 (name)
and
rem_2 == encrypted_sha256 (name)

Then the serial number is valid, else it’s not valid.

So how do we generate a valid serial:
- take name
- hash it using E2 and the already defined 128 bit key (look up)
- having name’s sha256 and encrypted sha256 find a number S so that
  S % prime1 == sha256
  S % prime2 == encrypted_sha256
  Because prime1 and prime2 have the greatest reminder 1 (duh, they are primes) this problem has a solution always. This is known as the Chinese Reminder Theorem. To find S we use Miracl bignums functions.
- Print S as a bignum in base16 as serial

One problem still ramins: if sha256 => prime1 or encrypted_sha256 => prime2 then we have no valid serial, as you all might know, the reminder is always smaller then the divisor.
Because of this I found out that many names don’t have a corresponding serial number: e.g.: bLaCk, bLaCk-eye, Veneta…

Final words:
It was a very easy crypto crackme, just for newbies.
Check the source of the keygen.

Greets:
Kanal23, TKM! , and RET reversing groups

Best wishes and have a nice new year
bLaCk-eye
mycherynos@yahoo.com