Solution to S!x0r's Crackme#1 by S!x0r

baderj (http://www.johannesbader.ch)

Dec. 8th 2014

This crackme was published December 7th, 2014. It is rated "3 - Getting harder". The description reads:

> First, sorry for my bad English my main language is German
>
> I have been created a keygenme, called Crackme#1 It is not so hard,but nothing for newbies. The difficulty is your choice.
>
> The Goal: Create a working keygen

In the first part of this solution I show how to reverse engineer the underlying math equation of this crackme. The second part then is all about solving the equation, this part is in large part copied from my solution for the crackme `bb_crackme#1` by `svan70`[1]

Part 1: Decompiling

I'm using IDA to disassemble the code. The author *S!x0r* gives a very nice hint in the comments of his crackme regarding IDA:

> No special bignum. With the IDA flirt signature called
>
> "RESIGSv014PUB RE-SIGS v0.14 PUBLIC by dihux"
>
> You can create a label.map for OllyDBG
>
> Sorry for my bad English

Given *RE-SIGS* [2] signatures for IDA the crackme is trivial to decompile.

Reading the Username and Code

The username and code are read with two calls to `GetDlgItemTextA`:

```
.text:004010B1 ; int __stdcall get_username_and_code(HWND hDlg)
.text:004010B1 get_username_and_code proc near         ; CODE XREF: DialogFunc+2Dp
.text:004010B1
.text:004010B1 hDlg= dword ptr  8
.text:004010B1
.text:004010B1 push    ebp
.text:004010B2 mov     ebp, esp
```

[1] http://johannesbader.ch/2014/09/crackmes-de-svan70s-bb__crackme1/]
[2] https://tuts4you.com/download.php?view.3407

```
.text:004010B4 push    edi
.text:004010B5 push    esi
.text:004010B6 push    200h
.text:004010BB push    offset username
.text:004010C0 call    RtlZeroMemory
.text:004010C5 push    200h
.text:004010CA push    offset md5x_string
.text:004010CF call    RtlZeroMemory
.text:004010D4 push    200h
.text:004010D9 push    offset code
.text:004010DE call    RtlZeroMemory
.text:004010E3 push    200h                             ; cchMax
.text:004010E8 push    offset username                  ; lpString
.text:004010ED push    65h                              ; nIDDlgItem
.text:004010EF push    [ebp+hDlg]                       ; hDlg
.text:004010F2 call    GetDlgItemTextA
.text:004010F7 mov     username_length, eax
.text:004010FC push    200h                             ; cchMax
.text:00401101 push    offset code                      ; lpString
.text:00401106 push    66h                              ; nIDDlgItem
.text:00401108 push    [ebp+hDlg]                       ; hDlg
.text:0040110B call    GetDlgItemTextA
.text:00401110 or      eax, eax
.text:00401112 jz      short loc_401124
.text:00401114 cmp     username_length, 0
.text:0040111B jz      short loc_401124
.text:0040111D mov     eax, 1
.text:00401122 jmp     short loc_401126                 ; eax = 0 -> username or code empty
.text:00401122                                          ; eax = 1 -> OK
.text:00401124 ; ---------------------------------------------------------------------------
.text:00401124
.text:00401124 loc_401124:                             ; CODE XREF: get_username_and_code+61j
.text:00401124                                          ; get_username_and_code+6Aj
.text:00401124 xor     eax, eax
.text:00401126
.text:00401126 loc_401126:                             ; CODE XREF: get_username_and_code+71j
.text:00401126 pop     esi                              ; eax = 0 -> username or code empty
.text:00401126                                          ; eax = 1 -> OK
.text:00401127 pop     edi
.text:00401128 leave
.text:00401129 retn    4
.text:00401129 get_username_and_code endp
```

Goodboy-Message

After the `username` and `code` are read, we enter a `validate` subroutine. If the routine returns 1, we get to see the goodboy message:

```
.text:0040105B call    get_username_and_code
.text:00401060 cmp     eax, 1                           ; 1 = OK
.text:00401063 jnz     short loc_40106A
.text:00401065 call    validate
.text:0040106A
```

```
.text:0040106A loc_40106A:                          ; CODE XREF: DialogFunc+35j
.text:0040106A cmp     eax, 1
.text:0040106D jnz     short loc_4010AB
.text:0040106F push    40h                          ; uType
.text:00401071 push    offset Caption               ; "Nice :)"
.text:00401076 push    offset Text                  ; "Valid Serial"
.text:0040107B push    [ebp+hWnd]                   ; hWnd
.text:0040107E call    MessageBoxA
.text:00401083 jmp     short loc_4010AB
```

ROT-1

This is the start of the `validate` routine:

```
.text:0040112C validate proc near                   ; CODE XREF: DialogFunc+37p
.text:0040112C
.text:0040112C code= dword ptr -0Ch
.text:0040112C power= dword ptr -8
.text:0040112C modulus= dword ptr -4
.text:0040112C
.text:0040112C push    ebp
.text:0040112D mov     ebp, esp
.text:0040112F add     esp, 0FFFFFFF4h
.text:00401132 push    edi
.text:00401133 push    esi
.text:00401134 lea     esi, username
.text:0040113A lea     edi, username
.text:00401140 jmp     short loc_401146
.text:00401142 ; ---------------------------------------------------------------------------
.text:00401142
.text:00401142 loc_401142:                          ; CODE XREF: validate+1Dj
.text:00401142 lodsb
.text:00401143 dec     al                           ; ROT-1(username)
.text:00401145 stosb
.text:00401146
.text:00401146 loc_401146:                          ; CODE XREF: validate+14j
.text:00401146 cmp     byte ptr [esi], 0
.text:00401149 jnz     short loc_401142
```

It replaces the `username` with $\text{ROT}_{-1}(\textit{username})$, i.e., each letter in the username is replaced with the preceding letter in the alphabet. For example, `sheldon` becomes `rgdkcnm`.

MD5 of Shifted Username

After the username is shifted one letter we get to these lines:

```
.text:0040114B call    MD5Init
.text:00401150 push    username_length
.text:00401156 push    offset username
.text:0040115B call    MD5Update
.text:00401160 call    MD5Final
.text:00401165 mov     dword ptr [eax], 30782153h ; overwrite 5 highest md5 bytes
```

```
.text:0040116B mov     byte ptr [eax+4], 72h
.text:0040116F push    offset md5
.text:00401174 push    16
.text:00401176 push    eax
.text:00401177 call    _HexEncode@12   ; HexEncode(x,x,x)
```

Here the signatures by *dihux* really begin to shine; all subroutines get nice speaking names. The code calculates the MD5 sum of the shifted username and places the result - 16 bytes - at `[eax]`. The 5 most significant bytes are then replace by the constant value `5321783072`. The result is then converted to a hex string with `HexEncode`.

Three Big Numbers

The crackme then initializes three big numbers. I called them m, n, and c (for reasons that will become clear later):

```
.text:0040117C call    _bnCreate@0     ; bnCreate()
.text:00401181 mov     [ebp+n], eax
.text:00401184 call    _bnCreate@0     ; bnCreate()
.text:00401189 mov     [ebp+c], eax
.text:0040118C call    _bnCreate@0     ; bnCreate()
.text:00401191 mov     [ebp+m], eax
.text:00401194 push    [ebp+c]
.text:00401197 push    offset code
.text:0040119C call    _Hex2bn@8       ; Hex2bn(x,x)
.text:004011A1 push    [ebp+m]
.text:004011A4 push    offset code
.text:004011A9 call    _Hex2bn@8       ; Hex2bn(x,x)
.text:004011AE push    200h
.text:004011B3 push    offset code
.text:004011B8 call    RtlZeroMemory
.text:004011BD push    [ebp+n]
.text:004011C0 push    offset aAd08d0361cc7fe ; "AD08D0361CC7FE8D1D3EAC5A68394C95"
.text:004011C5 call    _Hex2bn@8       ; Hex2bn(x,x)
```

The numbers m and c are initialized with the value of `Hex2bn(code)`, this means **the code is a number in hexadecimal notation**. The number n is initialized to `Hex2bn(0xAD08D0361CC7FE8D1D3EAC5A68394C95)`, which is 230002204674084418548395124071717227669.

Square-and-Multiply

Next we have:

```
.text:004011CA mov     edi, 0Fh                         ; square and multiply
.text:004011CF jmp     short loc_4011EE
.text:004011D1 ; ---------------------------------------------------------------------------
.text:004011D1
.text:004011D1 loc_4011D1:                              ; CODE XREF: validate+C4j
.text:004011D1 push    [ebp+c]
.text:004011D4 push    [ebp+c]
.text:004011D7 push    [ebp+c]
```

```
.text:004011DA call    _bnMul@12                        ; bnMul(x,x,x)
.text:004011DF push    [ebp+c]
.text:004011E2 push    [ebp+n]
.text:004011E5 push    [ebp+c]
.text:004011E8 call    _bnMod@12                        ; bnMod(x,x,x)
.text:004011ED dec     edi
.text:004011EE
.text:004011EE loc_4011EE:                              ; CODE XREF: validate+A3j
.text:004011EE or      edi, edi
.text:004011F0 jnz     short loc_4011D1
.text:004011F2 push    [ebp+c]
.text:004011F5 push    [ebp+c]
.text:004011F8 push    [ebp+c]
.text:004011FB call    _bnMul@12       ; bnMul(x,x,x)
.text:00401200 push    [ebp+c]
.text:00401203 push    [ebp+m]
.text:00401206 push    [ebp+c]
.text:00401209 call    _bnMul@12       ; bnMul(x,x,x)
.text:0040120E push    [ebp+c]
.text:00401211 push    [ebp+n]
.text:00401214 push    [ebp+c]
.text:00401217 call    _bnMod@12       ; bnMod(x,x,x)
```

These lines boil down to:

```
REPEAT 15 TIMES
    c = c*c
    c = c % n
END REPEAT
c = c*c
c = c*m
c = c % n
```

This is the *square-and-multiply* way to calculate:

$$c = m^{2^{16}+1} \bmod n$$

Check Result

There are only a couple of lines remaining in the `validate`-subroutine:

```
.text:0040121C push    offset code
.text:00401221 push    [ebp+c]
.text:00401224 call    _bn2Hex@8                        ; bn2Hex(x,x)
.text:00401229 push    [ebp+n]                          ; lpMem
.text:0040122C call    _bnDestroy@4                     ; bnDestroy(x)
.text:00401231 push    [ebp+c]                          ; lpMem
.text:00401234 call    _bnDestroy@4                     ; bnDestroy(x)
.text:00401239 push    [ebp+m]                          ; lpMem
.text:0040123C call    _bnDestroy@4                     ; bnDestroy(x)
.text:00401241 call    _bnFinish@0                      ; bnFinish()
.text:00401246 lea     esi, code
```

```
.text:0040124C lea     edi, md5
.text:00401252 push    edi                             ; lpString2
.text:00401253 push    esi                             ; lpString1
.text:00401254 call    lstrcmpA
.text:00401259 or      eax, eax
.text:0040125B jnz     short loc_401264
.text:0040125D mov     eax, 1
.text:00401262 jmp     short loc_401266
.text:00401264 ; ---------------------------------------------------------------------------
.text:00401264
.text:00401264 loc_401264:                             ; CODE XREF: validate+12Fj
.text:00401264 xor     eax, eax
.text:00401266
.text:00401266 loc_401266:                             ; CODE XREF: validate+136j
.text:00401266 pop     esi
.text:00401267 pop     edi
.text:00401268 leave
.text:00401269 retn
```

These lines first convert the variable `c` to a hex string, and store the result in `code`. The string `code` is then compared to the `md5` string. If they match, then the code returns 1 and we get the goodboy message. This means

$$c = m^{2^{16}+1} \bmod n \stackrel{!}{=} md5(\mathrm{ROT}_{-1}(username))$$

where `m` is the code that we enter.

Part 2: Solving the Equation

Solving the crackme is all about solving the following problem: given e, c and n, find m such that:

$$m^e \equiv c \bmod n$$

In other words, we need to find the eth root of c - which is hard in general. The exponent $e = 2^1 6 + 1 = 65537$ is a common choice for the public exponent in the RSA algorithm. This algorithm operates with moduli n that have two prime factors. Let's see if that is the case for our n. I'm using the free computer algebra system PARI/GP [3] to do the maths for me:

```
? factorint(230002204674084418548395124071717227669)
%1 =
[13603283776616498593 1]

[16907844344866863733 1]
```

Sure enough our n is a valid RSA modulus (except of course it has way to many bits to be secure - this is key to break the crackme). In the RSA asymmetric encryption, the ciphertext $c \equiv m^e \mod n$ can be decrypted to the plaintext message m using the private key d:

$$m \equiv c^d \mod n$$

[3] http://pari.math.u-bordeaux.fr

In our case the ciphertexts is the md5 sum of the ROT_{-1} of the username. The public key is $e = 65537$, and the modulus n is 230002204674084418548395124071717227669. If we can get the private key d we can calculate m.

The RSA Key Generation

The Wikipedia page on Key Generation [4] nicely shows how the public and private key are calculated:

Step 1 and 2 - $n = pq$ Choose two distinct primes p and q and determine the product n. We have n and need to determine its two prime factors p and q. The RSA algorithm is based on the fact that this is not feasible if n is large enough. Lucky for us, n is quite small in this crackme and we can get the two factors very fast (again I‘m using PARI/GP):

```
? n = 230002204674084418548395124071717227669;
? f = factorint(n);
? p = f[1,1]
%1 = 13603283776616498593
? q = f[2,1]
%2 = 16907844344866863733
```

So $p = 13603283776616498593$ and $q = 16907844344866863733$.

Step 3 - $\phi(n)$ Compute $\phi(n)$, where ϕ is the Euler‘s totient function. Because the primefactors of n are known, this is easy

```
? phi_n = (p-1)*(q-1)
%3 = 230002204674084418517883995950233865344
```

Step 4 - Chose the public key Choose an integer e such that $1 < e < \phi(n)$. Our e is given by the crackme: $e = 65537$ - which is a valid public key because it is smaller than $\phi(n)$.

```
? e = 2^16 + 1
%4 = 65537
```

Step 5 - Determine the private key Finally the interesting part. The private key is given by

$$d \equiv e^{-1} \mod \phi(n)$$

```
? d = (1/e) % phi_n
%5 = 35066939730281390814817536468479435777
```

Private-Key-Script

The following GP script performs the above steps to calculate the private key for this crackme:

[4] http://en.wikipedia.org/wiki/RSA_%28cryptosystem%29#Key_generation

```
rsa_private_key(e, n) = {
    /* e is the public key
    n is the modulus

    returns: private key d */

    /* factor n */
    f = factorint(n);

    /* check if m has exactly two prime factors */
    nrfacs = sum(i=1,matsize(f)[1], f[i,2]);
    if(nrfacs != 2, return(Str("n has ", nrfacs, " factors (not 2)!"));););

    /* get factors p*q = n */
    p = f[1,1];
    q = f[2,1];

    /* euler totient */
    phi_n = (p-1)*(q-1);

    /* make sure 1 < e < phi_n */
    if(e >= phi_n, return(Str("e is larger than phi(n)")););

    /* determine private key d as d = e^-1 mod phi_n */
    d = (1/e) % phi_n;
    return(d);

}

e = 2^16+1;         /* public key */

/* AD08D0361CC7FE8D1D3EAC5A68394C95 as decimal */
n = 230002204674084418548395124071717227669; /* modulus n=p*q with two distinct primes p and q */
d = rsa_private_key(e, n);
print("private key is: ", d)
quit()
```

Install Pari/GP with `apt-get install pari-gp`, then run the script with `gp -q private_key.gp`

```
$ gp -q private_key.gp
private key is: 35066939730281390814817536468479435777
```

The Keygenerator

Now that we have the private key we can decrypt all ciphertexts `c` (the md5 sum) to get the plaintext `m` (the code value) with

$$m \equiv c^d \mod n$$

The following Python script does that:

```
import hashlib
import argparse

def keygen(username):
    """ private key, see private_key.gp """
    d = 35066939730281390814817536468479435777

    """ modulus """
    n = 0xAD08D0361CC7FE8D1D3EAC5A68394C95

    def rsa_decrypt(c, d, n):
        """
            c: ciphertext
            d: private key
            n: modulus
        """
        return pow(c, d, n)

    def rotm1(p):
        c = "".join([chr(ord(x)-1) for x in p])
        return c

    shifted = rotm1(username)
    md5 = hashlib.md5(shifted).hexdigest()
    md5 = "5321783072" + md5[10:]
    c = int(md5,16)
    m = rsa_decrypt(c, d, n)
    code = hex(m).rstrip("L").lstrip("0x")
    return code

if __name__=="__main__":
    desc = "Keygen for S!x0r's Crackme#1"
    parser = argparse.ArgumentParser(description=desc)
    parser.add_argument("username")
    args = parser.parse_args()
    code = keygen(args.username)
    print("""your credentials are:
    username: {}
    code:     {}""".format(args.username, code))
```

For example:

```
$ python keygen.py baderj
your credentials are:
    username: baderj
    code:     97237b1f20f501b18a6ce54cf9fb7858
```

And you should see the good boy message 1.

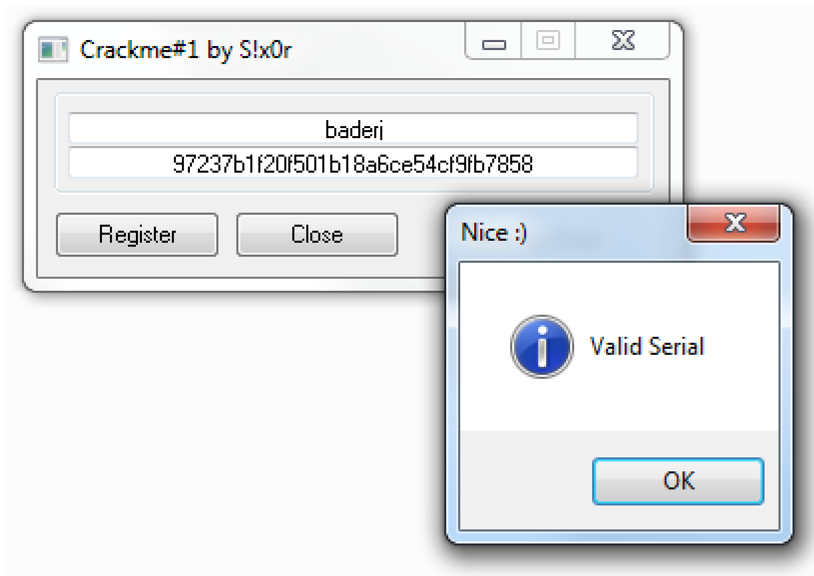

Figure 1: Goodboy Message