Solution to bb_crackm#1 by svan70

author: baderj (www.johannesbader.ch)

Crackme Infos:

- name: bb_crackme#1
- author: svan70
- published: 2014-06-16
- difficulty: 2 - Needs a little brain (or luck)
- plattform: Windows
- language: C/C++
- link: http://www.crackmes.de/users/svan70/bb_crackme1/
- description: Find please the correct code without patching. Little crypto knowledge is advantage. Good luck.

The crackme is very easy to disassemble and I'll therefore discuss the code only superficially in the first part of this solution. The main challenge of the crackme is the math required to produce the valid code. Jump to the second part if you are only interested in the crypto promised by the description of the crackme.

Part 1: The Disassembly

Extract Integer From Code

The first part of the disassembly is rather long::

```
.text:004010B1                 lea     edx, [esp+128h+code]
.text:004010B5                 push    edx             ; lParam
.text:004010B6                 push    100h            ; wParam
.text:004010BB                 push    0Dh             ; Msg
.text:004010BD                 push    3E8h            ; nIDDlgItem
.text:004010C2                 push    edi             ; hDlg
.text:004010C3                 mov     [esp+13Ch+var_10C], 120076516
.text:004010CB                 mov     [esp+13Ch+var_108], 1841478100
.text:004010D3                 mov     [esp+13Ch+var_104], 987884830
.text:004010DB                 call    ds:SendDlgItemMessageA
.text:004010E1                 test    eax, eax
.text:004010E3                 jz      short loc_401078
.text:004010E5                 mov     eax, dword_406330
.text:004010EA                 mov     ebp, 1
.text:004010EF                 cmp     eax, ebp
.text:004010F1                 jle     short loc_401109
.text:004010F3                 mov     eax, [esp+128h+code]
.text:004010F7                 push    4               ; Type
.text:004010F9                 and     eax, 0FFh
.text:004010FE                 push    eax             ; C
.text:004010FF                 call    __isctype
.text:00401104                 add     esp, 8
.text:00401107                 jmp     short loc_40111F
.text:00401109 ; ---------------------------------------------------------------------------
```

```
.text:00401109
.text:00401109 loc_401109:                             ; CODE XREF: DialogFunc+D1j
.text:00401109                 mov     ecx, [esp+128h+code]
.text:0040110D                 mov     edx, hardcoded
.text:00401113                 and     ecx, 0FFh
.text:00401119                 mov     al, [edx+ecx*2]
.text:0040111C                 and     eax, 4
.text:0040111F
.text:0040111F loc_40111F:                             ; CODE XREF: DialogFunc+E7j
.text:0040111F                 test    eax, eax
.text:00401121                 jnz     short loc_401131
.text:00401123                 pop     edi
.text:00401124                 mov     eax, ebp
.text:00401126                 pop     esi
.text:00401127                 pop     ebp
.text:00401128                 add     esp, 11Ch
.text:0040112E                 retn    10h
.text:00401131 ; ---------------------------------------------------------------------------
.text:00401131
.text:00401131 loc_401131:                             ; CODE XREF: DialogFunc+101j
.text:00401131                 mov     esi, ebp
.text:00401133
.text:00401133 loc_401133:                             ; CODE XREF: DialogFunc+145j
.text:00401133                 cmp     dword_406330, ebp
.text:00401139                 jle     short loc_40114E
.text:0040113B                 xor     eax, eax
.text:0040113D                 push    4               ; Type
.text:0040113F                 mov     al, byte ptr [esp+esi+12Ch+code]
.text:00401143                 push    eax             ; C
.text:00401144                 call    __isctype
.text:00401149                 add     esp, 8
.text:0040114C                 jmp     short loc_401160
.text:0040114E ; ---------------------------------------------------------------------------
.text:0040114E
.text:0040114E loc_40114E:                             ; CODE XREF: DialogFunc+119j
.text:0040114E                 mov     edx, hardcoded
.text:00401154                 xor     ecx, ecx
.text:00401156                 mov     cl, byte ptr [esp+esi+128h+code]
.text:0040115A                 mov     al, [edx+ecx*2]
.text:0040115D                 and     eax, 4
.text:00401160
.text:00401160 loc_401160:                             ; CODE XREF: DialogFunc+12Cj
.text:00401160                 test    eax, eax
.text:00401162                 jz      short loc_401167
.text:00401164                 inc     esi
.text:00401165                 jmp     short loc_401133
.text:00401167 ; ---------------------------------------------------------------------------
.text:00401167
.text:00401167 loc_401167:                             ; CODE XREF: DialogFunc+142j
.text:00401167                 cmp     byte ptr [esp+esi+128h+code], '-'
.text:0040116C                 jnz     loc_401326
.text:00401172                 lea     eax, [esp+128h+code]
```

The snippet checks if the code begins with a digit, if it doesn‘t, we failed the crackme. Next the code iterates over the characters in the code until it finds a non-digit. The snippet finally checks if this first non-digit is the hyphen character `-`:

```
i = 0
IF NOT isdigit(serial[i]) THEN
    RETURN FAIL

WHILE isdigit(serial[i]) DO
    i = i+1

IF serial[i] != "-" THEN
    RETURN FAIL
```

Extract and Convert

From the previous section we know that the code starts with `\d+-`, e.g., `18832-` or `7373-`. The next code segment uses two C standard library calls to parse the serial:

```
.text:00401172                 lea     eax, [esp+128h+code]
.text:00401176                 push    offset Delim    ; "-"
.text:0040117B                 push    eax             ; Str
.text:0040117C                 inc     esi
.text:0040117D                 call    _strtok
.text:00401182                 lea     ecx, [esp+130h+EndPtr]
.text:00401186                 push    0Ah             ; Radix
.text:00401188                 push    ecx             ; EndPtr
.text:00401189                 push    eax             ; Str
.text:0040118A           call   _strtoul     ; now eax holds first integer "1242-434" --> 1242d
.text:0040118F                 push    4073628529
.text:00401194                 push    eax
.text:00401195                 call    sub_401340
.text:0040119A                 mov     [esp+144h+var_118], eax
```

To library functions are:

- `_strtok`: splits the code text into tokens, separated by the `-` character.
- `_strtoul`: converts the first token string to an integer.

So if we enter the serial “1234-..” we get eax = 1234 in line `text:00401194`. The function call to `sub_401340` then takes this integer as the first argument, and a constant 4073628529 as the second argument.

Modular Exponentiation

Here is the subroutine at offset `0x00401340`. I renamed the first argument (the integer from the code) to `m`, and the second argument (the constant) to `n`:

```
.text:00401340 ; Attributes: bp-based frame
.text:00401340
```

```
.text:00401340 sub_401340      proc near               ; CODE XREF: DialogFunc+175p
.text:00401340                                         ; DialogFunc+21Fp ...
.text:00401340
.text:00401340 m                = dword ptr  8
.text:00401340 n                = dword ptr  0Ch
.text:00401340
.text:00401340                 push    ebp
.text:00401341                 mov     ebp, esp
.text:00401343                 mov     eax, [ebp+m]
.text:00401346                 mov     edx, [ebp+n]
.text:00401349                 cmp     eax, edx
.text:0040134B                 jnb     short loc_4013C1
.text:0040134D                 push    ebx
.text:0040134E                 mov     eax, [ebp+m]
.text:00401351                 mov     ebx, [ebp+n]
.text:00401354                 mul     eax
.text:00401356                 div     ebx
.text:00401358                 mov     eax, edx
.text:0040135A                 mul     eax
.text:0040135C                 div     ebx
.text:0040135E                 mov     eax, edx
.text:00401360                 mul     eax
.text:00401362                 div     ebx
.text:00401364                 mov     eax, edx
.text:00401366                 mul     eax
.text:00401368                 div     ebx
.text:0040136A                 mov     eax, edx
.text:0040136C                 mul     eax
.text:0040136E                 div     ebx
.text:00401370                 mov     eax, edx
.text:00401372                 mul     eax
.text:00401374                 div     ebx
.text:00401376                 mov     eax, edx
.text:00401378                 mul     eax
.text:0040137A                 div     ebx
.text:0040137C                 mov     eax, edx
.text:0040137E                 mul     eax
.text:00401380                 div     ebx
.text:00401382                 mov     eax, edx
.text:00401384                 mul     eax
.text:00401386                 div     ebx
.text:00401388                 mov     eax, edx
.text:0040138A                 mul     eax
.text:0040138C                 div     ebx
.text:0040138E                 mov     eax, edx
.text:00401390                 mul     eax
.text:00401392                 div     ebx
.text:00401394                 mov     eax, edx
.text:00401396                 mul     eax
.text:00401398                 div     ebx
.text:0040139A                 mov     eax, edx
.text:0040139C                 mul     eax
.text:0040139E                 div     ebx
```

```
.text:004013A0                 mov     eax, edx
.text:004013A2                 mul     eax
.text:004013A4                 div     ebx
.text:004013A6                 mov     eax, edx
.text:004013A8                 mul     eax
.text:004013AA                 div     ebx
.text:004013AC                 mov     eax, edx
.text:004013AE                 mul     eax
.text:004013B0                 div     ebx
.text:004013B2                 mov     eax, edx
.text:004013B4                 mov     ebx, [ebp+m]
.text:004013B7                 mul     ebx
.text:004013B9                 mov     ebx, [ebp+n]
.text:004013BC                 div     ebx
.text:004013BE                 mov     eax, edx
.text:004013C0                 pop     ebx
.text:004013C1
.text:004013C1 loc_4013C1:                             ; CODE XREF: sub_401340+Bj
.text:004013C1                 pop     ebp
.text:004013C2                 retn
.text:004013C2 sub_401340      endp
.text:004013C2
```

The code is long and repetitive, probably because the compiler applied (loop unwinding)[http://en.wikipedia.org/wiki/Loop_
The underlying algorithm is much shorter:

```
FUNCTION sub_401340(m, n)
    c = m
    IF c < n THEN
        REPEAT 16 TIMES:
            c = (c^2) % n
        c = (c*m) % n

    RETURN c
```

This is the (square-and-multiply)[http://en.wikipedia.org/wiki/Exponentiation_by_squaring] way to efficiently calculate

$$c = m^{2^{16}+1} \bmod 4073628529 = m^{65537} \bmod 4073628529$$

Those familiar with the RSA algorithm will notice that $2^{16}+1$ is a common choice for the public exponent e - more about that in the second part of this solution. The result of the subroutine is returned in `eax` and stored at `[esp+144h+var_118]` (line `.text:0040119A`).

Rinse and Repeat

The code segments discussed so far are repeated twice (the last time there is no check for the trailing `-`). This means our serial has the format `\d+-\d+-\d+`, for instance `727237-237-29389283`. For each of the three provided integers the code calculates

$$c_i = m_i^{65537} \bmod 4073628529$$

So for instance for the serial `727237-237-29389283`:

$$c_1 \equiv 727237^{65537} \equiv 29193 \bmod 4073628529$$
$$c_2 \equiv 237^{65537} \equiv 23875 \bmod 4073628529$$
$$c_3 \equiv 29389283^{65537} \equiv 39386 \bmod 4073628529$$

The Validation

After the three values have been calculated we enter this code segment::

```
.text:00401301 loc_401301:                             ; CODE XREF: DialogFunc+2F1j
.text:00401301                 mov     dl, byte ptr [esp+eax+128h+var_118]
.text:00401305                 mov     cl, byte ptr [esp+eax+128h+var_10C]
.text:00401309                 xor     dl, cl
.text:0040130B                 jnz     short loc_401326
.text:0040130D                 inc     eax
.text:0040130E                 cmp     eax, 0Ch
.text:00401311                 jl      short loc_401301
.text:00401313                 push    0               ; uType
.text:00401315                 push    offset aCongratulation ; "Congratulations!"
.text:0040131A           push   offset aYourCodeIsCorr ; "Your code is correct!\nPlease send your "...
.text:0040131F                 push    edi             ; hWnd
.text:00401320                 call    ds:MessageBoxA
```

`var_118` points to an integer array holding the three results; `var_10C` points to an array of three hardcoded integers set before:

```
.text:004010C3 mov     [esp+13Ch+var_10C], 120076516
.text:004010CB mov     [esp+13Ch+var_108], 1841478100
.text:004010D3 mov     [esp+13Ch+var_104], 987884830
```

The snippet loops 12 times and does a byte-wise comparison of three results to the hardcoded values. If they all match, we get the good boy message.

So we know that for a valid serial "$m_1 - m_2 - m_3$" the following conditions must all be true:

$$m_1^e \equiv 120076516 \mod n$$
$$m_2^e \equiv 1841478100 \mod n$$
$$m_3^e \equiv 987884830 \mod n$$

with $n = 4073628529$ and $e = 2^{16} + 1 = 65537$.

Part 2: The Math

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. But we already noticed that our $e = 65537$ 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)[http://pari.math.u-bordeaux.fr] to do the maths for me:

```
? factorint(4073628529)
%1 =
[47051 1]

[86579 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:

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

In our case the ciphertexts are the hardcoded integers (120076516,...), the public key is $e = 65537$, and the modulus n is 4073628529. If we can get the private key d we can calculate m.

The RSA Key Generation

(The Wikipedia page on Key Generation)[http://en.wikipedia.org/wiki/RSA_%28cryptosystem%29#Key_generation] 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 = 4073628529;
? f = factorint(4073628529);
? p = f[1,1]
%1 = 47051
? q = f[2,1]
%2 = 86579
```

So $p = 47051$ and $q = 86579$.

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 = 4073494900
```

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 $\phi(n) = 4073628529$.

```
? 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 = 3057436473
```

Decrypting the Ciphertext

Now that we have the private key we can decrypt all hardcoded messages, for instance for $c = 120076516$:

```
? m = lift(Mod(c,n)^d)
%6 = 580276954
```

The Keygenerator

The following PARI/GP Script calculates the private key d, decrypts the three hardcoded ciphertexts, and concatenates the result with "-" to get the one (and only) valid code:

```
/*
    1) Install Pari/GP with
            apt-get install pari-gp
    2) Run with
        gp -q keygen.gp
*/

rsa_decrypt(c, d, n) = {
    /* c is the cyphertext
    d is the private key
    n is the modulus

    returns plaintext m */
    m = lift(Mod(c,n)^d);
    return(m);
}

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 */
n = 4073628529;     /* modulus n=p*q with two distinct primes p and q */

d = rsa_private_key(e, n);
m1 = rsa_decrypt(120076516, d, n);
m2 = rsa_decrypt(1841478100, d, n);
m3 = rsa_decrypt(987884830, d, n);


print(Str(m1,"-",m2,"-",m3););
quit()
```

Running the script should produce:

```
$ gp -q keygen.gp
580276954-895936478-64598366
```