Solution to ksydfius's DCTF 4

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

24. Oct. 2014

The crackme focuses on cracking a crypto algorithm, rather than reverse engineering code. Hence, there is nothing special about the disassembly and I will only very briefly show how to decompile the algorithm in the first part of this solution. The second part then shows how to break the algorithm.

Reverse Engineering the Code

The Decryption Routine

The crackme calls the decryption routine in `sub_4013FA` three times on different ciphertexts. I renamed the subroutine to *decrypt*. It is called for the first time here:

```
.text:00401661 mov     eax, [ebp+key_len]
.text:00401667 mov     [esp+2F8h+key_len_], eax
.text:0040166B lea     eax, [ebp+key]
.text:00401671 mov     [esp+2F8h+key_], eax
.text:00401675 mov     eax, [ebp+var_220] ; 71h
.text:0040167B mov     [esp+2F8h+cipher_length], eax
.text:0040167F mov     [esp+2F8h+plaintext], ebx ; 28fbe0
.text:00401683 lea     eax, [ebp+cipherA]
.text:00401689 mov     [esp+2F8h+cipherA_], eax
.text:0040168C call    decrypt
```

The function prototype is:

```
void decrypt(char* cipher, char* plaintext, int cipher_length, char* key, int key_length)
```

The decryption routine `decrypt` has the following disassembly:

```
.text:004013FA ; =============== S U B R O U T I N E =======================================
.text:004013FA
.text:004013FA ; Attributes: bp-based frame
.text:004013FA
.text:004013FA decrypt proc near                           ; CODE XREF: sub_40151A+172p
.text:004013FA                                             ; sub_40151A+1A6p ...
.text:004013FA
.text:004013FA key_len_= dword ptr -14h
```

```
.text:004013FA i= dword ptr -10h
.text:004013FA s= dword ptr -0Ch
.text:004013FA key= dword ptr  8
.text:004013FA result= dword ptr  0Ch
.text:004013FA size= dword ptr  10h
.text:004013FA password= dword ptr  14h
.text:004013FA key_len= byte ptr  18h
.text:004013FA
.text:004013FA push    ebp
.text:004013FB mov     ebp, esp
.text:004013FD push    esi
.text:004013FE push    ebx
.text:004013FF sub     esp, 0Ch
.text:00401402 mov     [ebp+c], 1
.text:00401409 mov     [ebp+i], 0
.text:00401410
.text:00401410 loc_401410:                             ; CODE XREF: decrypt+61j
.text:00401410 mov     eax, [ebp+i]
.text:00401413 cmp     eax, [ebp+size]
.text:00401416 jge     short loc_40145D
.text:00401418 mov     eax, [ebp+result]
.text:0040141B mov     ecx, [ebp+i]
.text:0040141E add     ecx, eax
.text:00401420 mov     eax, [ebp+key]
.text:00401423 mov     ebx, [ebp+i]
.text:00401426 add     ebx, eax
.text:00401428 mov     edx, [ebp+i]
.text:0040142B lea     eax, [ebp+key_len]
.text:0040142E mov     [ebp+key_len_], eax
.text:00401431 mov     eax, edx
.text:00401433 mov     esi, [ebp+key_len_]
.text:00401436 cdq
.text:00401437 idiv    dword ptr [esi]                 ; i/password_length
.text:00401439 mov     eax, [ebp+password]
.text:0040143C movsx   eax, byte ptr [eax+edx]        ; password[i % password_length]
.text:00401440 imul    eax, [ebp+s]                    ; eax = eax * s
.text:00401444 xor     al, [ebx]                       ; eax XOR key[i]
.text:00401446 mov     [ecx], al              ; dst[i] = password[i%pw_len]*s ^ key[i]
.text:00401448 mov     eax, [ebp+key]
.text:0040144B add     eax, [ebp+i]
.text:0040144E movzx   edx, byte ptr [eax]
.text:00401451 lea     eax, [ebp+c]
.text:00401454 add     [eax], edx                      ; s += key[i]
.text:00401456 lea     eax, [ebp+i]
.text:00401459 inc     dword ptr [eax]                 ; i = i + 1
.text:0040145B jmp     short loc_401410
.text:0040145D ; ---------------------------------------------------------------------------
.text:0040145D
.text:0040145D loc_40145D:                             ; CODE XREF: decrypt+1Cj
.text:0040145D add     esp, 0Ch
.text:00401460 pop     ebx
.text:00401461 pop     esi
```

```
.text:00401462 pop     ebp
.text:00401463 retn
.text:00401463 decrypt endp
```

The code implements the following simple crypto algorithm:

```
FUNCTION decrypt(cipher, key)
    plaintext = char[cipher_len]
    s = 1
    FOR i in 0 TO len(cipher)-1 DO
        plaintext[i] = (key[i % len(key)]*s) ^ cipher[i] % 255
        s = s + cipher[i]
    END FOR
END
```

The Hash-Routine

The crackme uses a second routine `sub_401464` (renamed to `hash`) that validates the plaintext. It is called like this:

```
.text:004016C5 mov     byte ptr [ebx+70h], 0
.text:004016C9 mov     [ebp+var_22F], 0
.text:004016D0 mov     [esp+2F8h+plaintext_], ebx      ; plaintext from decrypt
.text:004016D3 call    hash
```

The function prototype is:

```
int hash(char* plaintext)
```

It has the following disassembly:

```
:text.0401464 ; int __cdecl hash(char *plaintext)
.text:00401464 hash proc near                          ; CODE XREF: sub_40151A+1B9p
.text:00401464                                         ; sub_40151A+1C9p ...
.text:00401464
.text:00401464 result1= dword ptr -18h
.text:00401464 plaintext_length_= dword ptr -10h
.text:00401464 i= dword ptr -0Ch
.text:00401464 plaintext_length= dword ptr -8
.text:00401464 c= dword ptr -4
.text:00401464 plaintext= dword ptr  8
.text:00401464
.text:00401464 push    ebp
.text:00401465 mov     ebp, esp
.text:00401467 sub     esp, 18h
.text:0040146A mov     [ebp+c], 1
.text:00401471 mov     eax, [ebp+plaintext]            ; result1 28fbe0
.text:00401474 mov     [esp+18h+result1], eax          ; Str
.text:00401477 call    strlen                          ; should be 0x70 for result1
```

```
.text:0040147C mov     [ebp+plaintext_length], eax
.text:0040147F mov     [ebp+i], 0
.text:00401486
.text:00401486 loc_401486:                             ; CODE XREF: hash+67j
.text:00401486 mov     eax, [ebp+i]
.text:00401489 cmp     eax, [ebp+plaintext_length]
.text:0040148C jge     short loc_4014CD
.text:0040148E mov     eax, [ebp+plaintext]
.text:00401491 add     eax, [ebp+i]
.text:00401494 movsx   edx, byte ptr [eax]             ; res[i]
.text:00401497 mov     eax, [ebp+c]                    ; one at start
.text:0040149A imul    eax, edx                        ; res[i]*fac
.text:0040149D mov     [ebp+c], eax                    ; fac = res[i]*fac
.text:004014A0 mov     edx, [ebp+i]
.text:004014A3 inc     edx
.text:004014A4 lea     eax, [ebp+plaintext_length]
.text:004014A7 mov     [ebp+plaintext_length_], eax
.text:004014AA mov     eax, edx                        ; eax = i+1
.text:004014AC mov     ecx, [ebp+plaintext_length_]
.text:004014AF cdq
.text:004014B0 idiv    dword ptr [ecx]                 ; edx = (i+1) % res_len
.text:004014B2 mov     eax, [ebp+plaintext]
.text:004014B5 movsx   edx, byte ptr [eax+edx]        ; edx = result[(i+1) % res_len]
.text:004014B9 lea     eax, [ebp+c]
.text:004014BC add     [eax], edx                      ; fac = fac + edx
.text:004014BE mov     edx, [ebp+i]
.text:004014C1 lea     eax, [ebp+c]
.text:004014C4 xor     [eax], edx                      ; fac = fac ^ i
.text:004014C6 lea     eax, [ebp+i]
.text:004014C9 inc     dword ptr [eax]
.text:004014CB jmp     short loc_401486
.text:004014CD ; ---------------------------------------------------------------------------
.text:004014CD
.text:004014CD loc_4014CD:                             ; CODE XREF: hash+28j
.text:004014CD mov     eax, [ebp+c]
.text:004014D0 leave
.text:004014D1 retn
.text:004014D1 hash endp
.text:004014D1
.text:004014D2 ; ---------------------------------------------------------------------------
.text:004014D2 push    ebp
.text:004014D3 mov     ebp, esp
.text:004014D5 sub     esp, 8
.text:004014D8 mov     dword ptr [ebp-4], 0
.text:004014DF
.text:004014DF loc_4014DF:                             ; CODE XREF: .text:0040150Cj
.text:004014DF mov     eax, [ebp-4]
.text:004014E2 cmp     eax, [ebp+10h]
.text:004014E5 jge     short loc_40150E
.text:004014E7 mov     eax, [ebp+8]
.text:004014EA mov     ecx, [ebp-4]
.text:004014ED add     ecx, eax
```

```
.text:004014EF mov     eax, [ebp+0Ch]
.text:004014F2 mov     edx, [ebp-4]
.text:004014F5 add     edx, eax
.text:004014F7 movzx   eax, byte ptr [ecx]
.text:004014FA cmp     al, [edx]
.text:004014FC jz      short loc_401507
.text:004014FE mov     dword ptr [ebp-8], 0
.text:00401505 jmp     short loc_401515
.text:00401507 ; ---------------------------------------------------------------------------
.text:00401507
.text:00401507 loc_401507:                             ; CODE XREF: .text:004014FCj
.text:00401507 lea     eax, [ebp-4]
.text:0040150A inc     dword ptr [eax]
.text:0040150C jmp     short loc_4014DF
.text:0040150E ; ---------------------------------------------------------------------------
.text:0040150E
.text:0040150E loc_40150E:                             ; CODE XREF: .text:004014E5j
.text:0040150E mov     dword ptr [ebp-8], 1
.text:00401515
.text:00401515 loc_401515:                             ; CODE XREF: .text:00401505j
.text:00401515 mov     eax, [ebp-8]
.text:00401518 leave
.text:00401519 retn
```

It boils down to this algorithm:

```
FUNCTION hash(plaintext)
    DWORD hash = 1
    FOR i in 0 TO len(plaintext)-1 DO
        ps = SIGNED plaintext[i]
        psp = SIGNED plaintext[(i+1) % len(plaintext)]
        hash = ((hash*ps) + psp) ^ i
    END FOR
    RETURN hash
END
```

The crackme uses hashes calculated with the above routine to check if the plaintext is correct. I didn't use the algorithm to crack the code though.

Cracking the Decryption Algorithm

As seen in the previous section, the decryption algorithm is:

```
FUNCTION decrypt(cipher, key)
    plaintext = char[cipher_len]
    s = 1
    FOR i in 0 TO len(cipher)-1 DO
        plaintext[i] = (key[i % len(key)]*s) ^ cipher[i] % 255
        s = s + cipher[i]
    END FOR
END
```

Let n be the length of the plaintext and ciphertext. Furthermore, let the plaintext and ciphertext be $p_1 \dots p_n$ and $c_1 \dots c_n$ respectively. The variables p_i and c_i denote the ith byte of the plaintext and ciphertext. Let the key be $k_1 \dots k_l$, where l is the length of the key and k_i is the ith byte of the key. Then we have the following relation between plaintext and ciphertext:

$$\begin{aligned}
p_1 &= c_1 \oplus k_1 \cdot 1 \\
p_2 &= c_2 \oplus k_{2 \bmod l} \cdot (1 + c_1) \\
p_3 &= c_3 \oplus k_{3 \bmod l} \cdot (1 + c_1 + c_2) \\
&\vdots \\
p_n &= c_n \oplus k_{n \bmod l} \cdot \left(1 + \sum_{i=1}^{n-1} c_n\right)
\end{aligned}$$

So how do we find the key k_i? First, notice that k_i only affects bytes j, with $j == i \mod l$. This also means that the ith plaintext byte only depends on the ciphertext (up to byte i), and the key byte $k_{i \bmod l}$:

$$\left(c_i, \sum_{j=0}^{i-1} c_i, k_{i \bmod l}\right) \mapsto p_i$$

So for instance in a key with length 8, the plaintext characters $p_2, p_{10}, p_{18}, \dots$ depend all on k_2, and k_2 only. This means we can crack one character of the key at a time.

What do we know about the plaintext? Not much, we only know that the last of the three decrypted plaintext messages will be printed to stdout::

```
00401789 mov      [esp+2F8h+plaintext_], offset aS ; "\n= %s =\n
```

So at least the third plaintext should be in ASCII, probably an English sentence. I'm assuming the same holds true for the first two plaintexts. The key should therefore produce plaintexts which have the expected character distribution of English text. I used the first distribution than came up in a Google search: `http://fitaly.com/board/domper3/posts/136.html`. Let $f(a)$ be the frequency of character a. Using this distribution, we can find the best guess for a key byte k_i, given the key length l (which we don't know). Let κ_i^l denote the best guess for the ith character of the key, given the length l:

$$\kappa_i^{\mathrm{l}} = \operatorname*{argmax}_{0 \le v \le 256} \prod_{j=i}^{\lfloor \frac{n}{l} \rfloor} f\left(c_{jl} \oplus v \cdot \sum_{d=i}^{j-1} c_{dl}\right)$$

In other words, for all potential key values v, we calculate the resulting cipher text and take the product of the probabilities of a given cipher text bytes. We then take the v that produces the larges product of probabilities. Since we don't know the key length, we also need to iterate over all potential key lengths. By summing up the scores of all the best guesses of the key characters, and picking the best one, we find the most probable key length λ:

$$\lambda = \operatorname*{argmax}_{0 \le l \le n} \left(\sum_{i=0}^{l} \max_{0 \le v \le 256} \prod_{j=i}^{\lfloor \frac{n}{l} \rfloor} f\left(c_{jl} \oplus v \cdot \sum_{d=i}^{j-1} c_{dl}\right)\right)$$

The following Python script implements these equations to find the most probable key length, with the corresponding most probable key bytes::

```
import struct
import re

def get_freq():
    frequencies = {}
    with open("ascii_frequencies.txt", "r") as r:
        for f in r:
            m = re.search("(\d+)\s.*\(\s*([\d.]+)", f)
            if m:
                frequencies[int(m.group(1))] = float(m.group(2))

    return frequencies

def read_file(path):
    data = []
    with open(path, 'rb') as r:
        while True:
            dat = r.read(1)
            if dat != "" and len(dat):
                data.append(struct.unpack('B',dat)[0])
            else:
                return data


def decrypt(cipher, key):
    result = len(cipher)*[None]
    varc = 1
    for i, c in enumerate(cipher):
        result[i] = ( (key[i % len(key)]*varc) ^ c ) & 0xFF
        varc += c
    return result


def list_to_string(l):
    return "".join([chr(c) if 32 <= c <= 126 else "?" for c in l])

def best_key(cipher, weights, l, i, freq):
    wa = weights[i::l]
    ca = cipher[i::l]
    best_score = -1
    best_key = None
    for k in range(32, 126):
        tmp = []
        for w, c in zip(wa, ca):
            tmp.append(((w*k) ^ c) & 0xFF)
        score = 1
        for x in tmp:
            score *= freq.get(int(x),0)
        if score > best_score:
```

```
            best_score = score
            best_key = k
    return best_key, best_score

def crack(cipher):
    freq = get_freq()
    weights = len(cipher)*[None]
    s = 1
    for i, c in enumerate(cipher):
        weights[i] = s
        s += c
        s = s & 0xFF

    overall_best_score = 0
    overall_best_key = None
    for key_len in range(2,80):
        score_sum = 0
        key = key_len*[None]
        for i in range(key_len):
            key[i], score = best_key(cipher, weights, key_len, i, freq)
            score_sum += score

        if score_sum > overall_best_score:
            overall_best_score = score_sum
            overall_best_key = key
    return overall_best_key


for nr in ["A", "B", "C"]:
    cipher = read_file("cipher" + nr)
    key = crack(cipher)
    msg = decrypt(cipher, key)
    print("cipher {}\n========\nkey: {}\nplaintxt: {}\n\n".format(nr,
        list_to_string(key), list_to_string(msg)))

key_txt = "SU5sc0tFU0Rhp4Jziu0pfHspW"
print("The best key is obviously {}".format(key_txt))
key = [ord(k) for k in key_txt]


for nr in ["A", "B", "C"]:
    cipher = read_file("cipher" + nr)
    msg = decrypt(cipher, key)
    print("cipher {}\n========\nplaintxt: {}\n\n".format(nr,
        list_to_string(msg)))
```

The first part of the output is::

```
cipher A
========
key: SU5sc0tFU0Rhp4Jziu0pfHspW
plaintxt: This is a very weak algorithm and should only be used for educational purposes. So, I challenge y
```

```
cipher B
========
key: SU5sc0tFUORhp4J5iuOpfHsOW
plaintxt: How are you doi g so faM? You seem to ha,e made trogress since you can re d this message.


cipher C
========
key: evOH'E-Mg%5.}0Mj
plaintxt: sXitc-reHMRTtdpiOo _To*s1*GaNsQ.fiadfll n(r3USS
```

The script found the correct key given the first cipher text message. It almost got it right for the second ciphertext. The third ciphertext (the shortest of the three) was not correctly decrypted. But since the key is the same for all three message, we can use the correct key from the first ciphertext and apply it two all three ciphertexts - this is what the second half of the script output shows::

```
The best key is obviously SU5sc0tFUORhp4JziuOpfHspW
cipher A
========
plaintxt: This is a very weak algorithm and should only be used for educational purposes. So, I challenge y


cipher B
========
plaintxt: How are you doing so far? You seem to have made progress since you can read this message.


cipher C
========
plaintxt: Easy? Submit this: _W34kAlG0R1ThM5aR3b4dF0rY0U_
```

If we enter the correct key “SU5sc0tFUORhp4JziuOpfHspW” to the crackme we get the flag:

```
>dctf4_final.exe
===============
=   DEFCAMP   =
=  REVERSING  =
=      4      =
===============
Password: SU5sc0tFUORhp4JziuOpfHspW

===================================================
= Easy? Submit this: _W34kAlG0R1ThM5aR3b4dF0rY0U_ =
===================================================
```