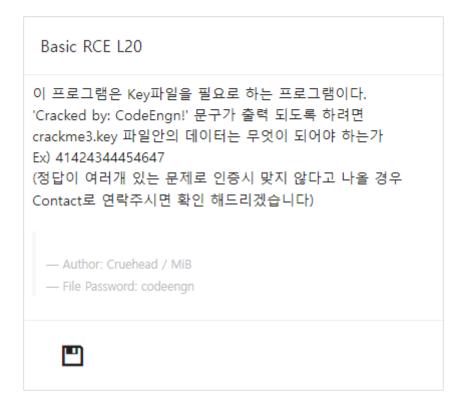
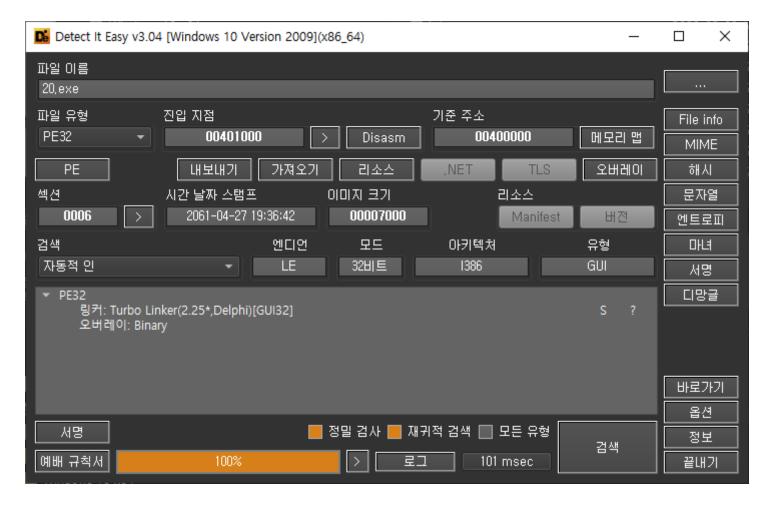
codeengn-basic-L20 풀이

리버싱 문제풀이 / Wonlf / 2022. 4. 28. 14:53



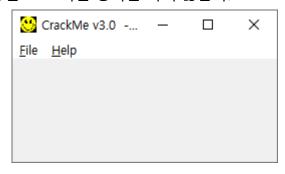
문제는 Key파일을 구하고, 프로그램이 실행 될 때, CodeEngn! 문구가 출력 되어야 한다고 한다.

Die로 열어보면,



특이사항은 보이지 않는다.

프로그램을 실행해보면, 이러한 창만 뜨고 다른 동작은 하지 않는다.



디버거로 열어본다.

```
00401016
00401018
                       80000000
                    68
                                                         80
                                                   push
● 0040101D
                        03
                                                   push
0040101F
                    6A
                       00
                                                   push
                    6A
68
                                                   push
00401021
                       03
00401023
                        000000C0
                                                   push C0000000
                                                   push 20.402007
call <JMP.&CreateFileA>
cmp eax,FFFFFFFF
jne 20.401043
push 20.40210E
00401028
                    68
                       D7204000
76040000
                                                                                                                  4020D7: "CRACKME3.KEY"
  0040102D
                    E8
                   83F8 FF
75 OC
68 0E214000
E8 B4020000
00401032
                                                   push 20.40210E

call <20.sub_4012F5>

jmp 20.4010AE
00401037
                                                                                                                  40210E: "CrackMe v3.0
00401030
00401041
                   EB 6B
A3 F52
                       F5204000
12000000
                                                   mov dword ptr ds:[4020F5],eax
                    B8
                                                   mov eax,12
mov ebx,20.402008
00401048
● 0040104D
                        08204000
                                                                                                                  ebx: "abcdefghijklmnopqr", 402008: "abcdefghijklmnopqr"
00401052
                    6A
00401054
                        A0214000
                                                   push 20.4021A0
                    50
                                                   push eax
  004010
                                                                                                                  ebx: "abcdefghijklmnopqr"
                   FF35 F5204000
E8 30040000
833D A0214000 12
75 C8
68 08204000
E8 98020000
                                                  push dword ptr ds:[4020F5]
call <JMP.&ReadFile>
cmp dword ptr ds:[4021A0],12
                                                                                                                  파일 읽기
길이를 비교하는 구문 길이를 0x12(18)로 정해보자
00401066
0040106F
                                                                                                                  402008: "abcdefghijklmnopqr"
                                                   push 20.402008
call <20.sub_401311:
00401074
```

한줄씩 실행시키다 보면, CreateFileA과 ReadFile로 파일을 읽고 있다. "CRACKME3.KEY" 라는 파일을 읽어서 그 안에 있는 내용이 0x12(18)길이가 아니면 다른 곳으로 점프를 한다.

401311함수를 실행시켜야 정상 구문으로 진행 될 거 같으니 "CRACKME3.KEY" 파일을 생성하고 파일 내용을 a~r까지 총 18자리로 구성하고 디버깅을 하였다.

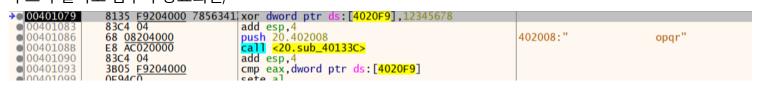
18자리 검증을 통과하고 401311함수로 들어와서 확인해보면,

```
ecx 0 만들고
eax 0 만들고
                 33C9
00401311
                                          xor ecx,ecx
                 33C0
                                          xor eax, eax
                8B7424 04
                                          mov esi,dword ptr ss:[esp+4]
mov bl,41
  00401315
                                                                                               KEY안에 데이터를 esi에 넣는다
  00401319
                 B3 41
                                                                                               41: A
esi:EntryPoint
입력 받은 값의 첫번째 1바이트와 0x41을 xor하여 eax에 넣음
xor한 값과 앞 1바이트를 교체
esi:EntryPoint
                                          mov al, byte ptr ds:[esi]
  0040131B
                 8A06
  0040131D
                 32C3
                                               al.bl
  0040131F
                 8806
                                          mov byte ptr ds:[esi],al
  00401321
                                          inc esi
                FFC3
  00401322
  00401324
                0105 F9204000
                                          add dword ptr ds:[4020F9],eax
                3C 00
74 07
  0040132A
                                          cmp_al,0
                                                                                               xor한 값을 O과 비교함
                                             20.401335
  0040132C
                 FEC1
                                          inc c
                                          cmp bl,4F
ine 20.40131B
00401330
                                                                                               0x41와 0x4F를 비교함
                 75 E6
                890D 49214000
                                          mov dword ptr ds:[402149],ecx
```

이 구문들을 해석해보면, (KEY[0] ^ 0x41) + (KEY[1] ^ 0x42) + (KEY[2] ^ 0x43).... 이런식으로 진행되는데, 0x4E까지 실행된다. 구문이 끝나고 inc bl이 되고 비교하니까 0x4F가 아닌 0x4E까지만 도는 것이다.

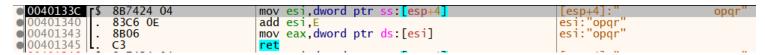
즉, KEY[0] ~ KEY[13](14자리)를 0x41~0x4E에 대응하여 xor하고 전부 더한 값이 4020F9에 담긴다.

루프가 끝나고 함수가 종료되면



4020F9에 담긴 값과 0x12345678을 xor을 한다.

그리고 18자리 중, 전에 연산했던 14자리를 뺀 나머지 4자리를 가지고 40133C함수를 호출하는데, 들어가보면



앞의 공백을 제거한다.

그리고 함수를 탈출하면

```
00401090
                                            add esp.
                 3B05 F9204000
                                            cmp eax, dword ptr ds:[4020F9]
  00401093
                 0F94C0
00401099
                                            sete al
0040109C
                 50
                                            push eax
● 0040109D
                 84C0
                                            test al, al
                                           je 20.401037
push 20.40210E
0040109F
                 74 96
                 68 <u>0E214000</u>
E8 <u>9B020000</u>
004010A1
                                                                                                  40210E: "CrackMe v3.0
                                            call <20.sub_401346>
004010A6
● 004010AB
                 83C4 04
                                            add esp,4
● 004010AE
                 6A 00
                                            push 0
                 68 <u>28214000</u>
E8 <u>9</u>A030000
● 004010B0
                                            push 20.402128
call <JMP.&FindWindowA>
                                                                                                  402128: "No need to disasm the code!"
004010B5
● 004010BA
                 0BC0
                                            or eax,eax
004010BC
                74 01
                                            je 20.4010BF
● 004010BE
                 C3
```

앞서 했던 연산의 결과인 4020F9와 18자리의 KEY중 뒷 4자리를 비교한다.

```
eax=7271706F
dword ptr ds:[20.004020F9]=123457B8
```

실제로 뒷 4자리인 "opqr"의 16진수인 6F707172가 리틀 엔디언으로 인해 역순으로 저장되어 비교되고 있다. 비교하는 값인 123457B8이 저장했던 18자리의 문자열 중 뒷 4자리여야 아래 구문으로 갈 수 있을 것이다.

이것을 토대로 앞 14자리를 입력했을 때, 대응되는 뒷 4자리를 구하는 코드를 작성해보겠다.

```
#include <stdio.h>

#include <stdio.h>

pint main(void) {
    char value[18] = "abcdefghijklmnopqr"; //18자리
    int temp = 0;

for (int i = 0x41, j = 0; i <= 0x4E; i++, j++) { //여차피 여기서 14번만 돌음
    temp = temp + (value[j] ^ i);
}

temp = temp ^ 0x12345678;

printf( format: "0x%X", temp);

printf( format: "0x%X", temp);

#include <stdio.h>

#include <stdio.h>
#include <stdio.h>

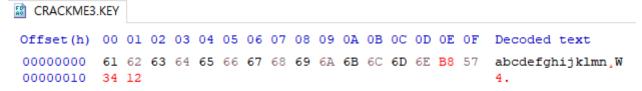
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>
#include <stdio.h>
#include <stdio.h

#include <stdio.h>
#include <stdio.h

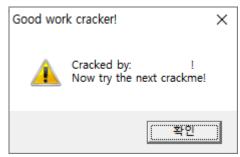
#include <std
```

코드로 직접 확인해보았다. 18자리중 앞 14자리가 "abcdefghijklmn" 이라면 뒤 4자리는 0x12, 0x34, 0x56, 0xB8이 되어야 한다. KEY를 수정해보자.

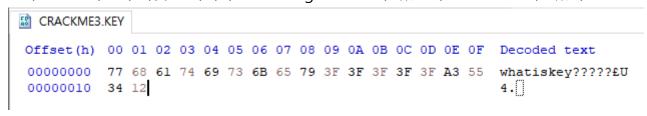
메모장으로는 16진수를 입력할 수 없으니, 16진수를 입력할 수 있는 HxD로 파일을 수정해주겠다.



리틀 엔디언 방식이니 역순으로 입력해주고 저장해서 프로그램을 다시 실행해보면,



이러한 알림창이 뜨게 되는데, 문제에서 말한 CodeEngn이 들어가야 할 칸은 저기 인 것 같다. 현재는 공백으로 보이는데 이것을 어떡하면 CodeEngn을 넣을 수 있는지 KEY를 변경해보았다.



0x123455A3

앞 14자리는 "whatiskey?????"이렇게 하고 뒷 4자리는 C로 짰던 코드를 사용해서 알아낸 뒤 넣고 프로그램을 실행 시키면,



이러한 문자열이 삽입 되는데, 총 14자리이고 확인해보면 이 문자열은 401311함수에 있던 처음 14자리를 0x41~0x4E까지 하나씩 xor한 값이 출력 되었다. 이제 생각해보면,

```
n ^0x41 = C

n1 ^0x42 = 0

n2 ^0x43 = d
```

이런식으로 CodeEngn이 출력되어야 할 때 n을 구할 수 있을 것이다.

C로 코드를 짜보면,

```
int main(void) {
    char value[8] = "CodeEngn";
    for (int i = 0x41, j = 0; i <= 0x4E; i++, j++) {
        if(j == 8){
            break;
        }
        printf( format: "0x%X ", value[j] ^ i);
    }
    test x

C:\Users\a02\Desktop\project\test\cmake-build-debug\test.exe
    0x2 0x2D 0x27 0x21 0x0 0x28 0x20 0x26</pre>
```

일단 KEY에 들어갈 14자리 중 CodeEngn에 대응하는 8자리 값은 찾았다.

뒷 6자리는 0으로 채우고 나머지 4자리를 알아보면,

```
#include <stdio.h>

int main(void) {
    int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [12]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [12]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [12]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [12]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [2]: 0x27, [3]: 0x21, [4]: 0x00, [5]: 0x28, [6]: 0x20, [7]: 0x26, [8]: 0x00, [9]: 0x00, [10]: 0x00, [11]: 0x00, [12]: 0x00, [13]: 0x00};

int key[] = { [0]: 0x02, [1]: 0x20, [1]: 0x20, [1]: 0x20, [1]: 0x00, [1]: 0x0
```

0x123452B0 이 되고

```
CRACKME3.KEY

Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text

000000000 02 2D 27 21 00 28 20 26 00 00 00 00 00 B0 52 .-'!.( &......°R

00000010 34 12 4.
```

이것을 HxD로 수정해서 프로그램을 다시 실행시켜 보면,



CodeEngn은 정상적으로 출력 되지만, 뒤 6자리가 IJKLMN이 붙어서 나온다. 이것을 지워보도록 하자.

처음부터 다시 생각해보면, KEY의 14자리를 0x41 ~ 0x4E까지 xor을 했다. 그럼 0x41 ~ 0x48 까지는 특정 값과 정상적으로 xor 이 되어 "CodeEngn" 이라는 결과가 출력 되었고, 0x49 ~ 0x4E는 0으로 채운 부분과 xor을 했기 때문에

"IJKLMN"이라는 결과가 출력 되었다.

XOR의 특징은 같은 값을 XOR 했을 때, 0을 반환하는 성질이 있다.

그렇다면, 0으로 채운 6자리 부분을 0x49 ~ 0x4E로 채우면 xor결과가 0이 되므로 아무것도 출력 하지 않을 것이다.

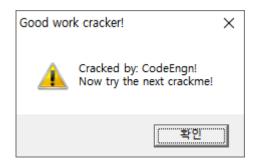
코드로 확인해보자.

앞 14자리를 "CodeEngn"에 대응하는 8자리와 나머지 6자리를 0x49 ~ 0x4E로 채웠을 때 맨 뒤 4자리는 0x1234557B이다.

이것을 HxD로 수정하고,

```
Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Decoded text
00000000 02 2D 27 21 00 28 20 26 49 4A 4B 4C 4D 4E 7B 55 .-'!.( &IJKLMN{U 00000010 34 12 4.[
```

프로그램을 실행해보자



이렇게 뒤에 "IJKLMN"문자열이 제거 된 모습이다. HxD에 있는 내용을 그대로 Auth에 인증하자.

정답!