От переводчика - свои объяснения и примечания я буду выделять курсивом.

1. BadBoy-функция - это код, сообщающий пользователю badboy-сообщение. badboy-сообщение - это сообщение, которое отображается пользователю в случае неверной пары имя-ключ. Текст сообщения разный, содержание всегда одно: "Вы ввели неправильное имя или пароль (ключ)". Либо другой вариант - пользователю ничего не сообщается, а программа ожидает ввода правильного ключа.
2. GoodBoy-функция - это код, сообщающий пользователю goodboy-сообщение. goodboy-сообщение - это ура-сообщение, которое показывается пользователю в случае корректной пары имя-ключ.

Crack-me для конференции BlackHat US 2012

Платформа: Windows
Язык: C/C++
Дано: Crack-me (http://solutionmes.wikidot.com/cm-eset-bh12)

Решение, автор: Eloi Vanderbeken (https://twitter.com/elvanderb @elvanderb)
Enthusiast reverse engineer of obfuscated and protected binary for 7 years. Currently working for Oppida. France
опубликовано 06.08.2012 (http://blog-oppida.blogspot.fr/2012/08/solution-for-eset-blackhat-us-challenge.html)

1. Введение

21 июля 2012 года в 7 утра (время по Лас-Вегасу) **ESET** (http://eset.com/) опубликовала **crack-me** для конференции BlackHat US 2012. Правила просты: найти правильную пару ключей для выбранного имени до 14:00 26 июля 2012 и выиграть $1000 и BlackCard (бесплатный билет на любое следующее мероприятие – BlackHat US 2013 или BlackHat EU 2013).

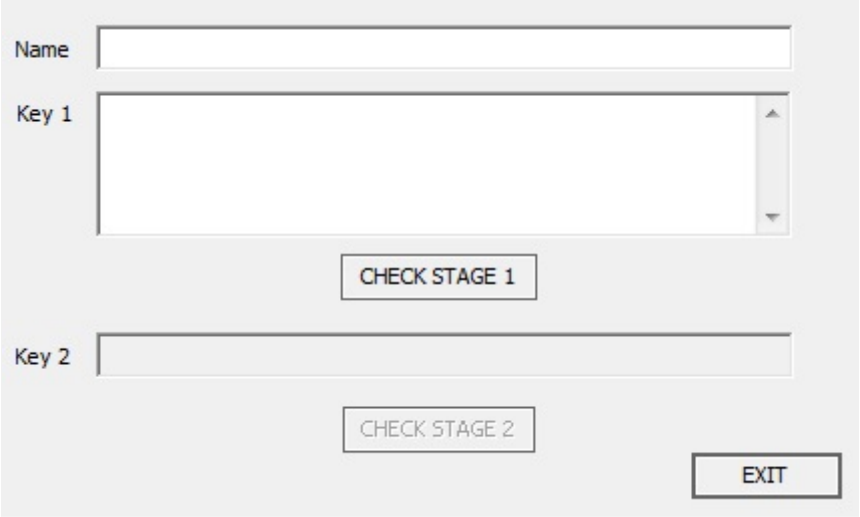


Рис. 1 Интерфейс crack-me

На страничке http://go.eset.com/us/rulethecode/ сообщается (учтите, что сама страничка может измениться), что Eloi Vanderbeken из Oppida был единственным, кто решил задачку в течении указанного времени и выиграл приз, при этом, он даже не присутствовал на BlackHat US:

Рис. 2 Поздравление победителю

В блоге по адресу http://blog-oppida.blogspot.com/2012/08/solution-for-eset-blackhat-us-challenge.html объясняется, как Eloi решил задачу.

2. Первые шаги

Crack-me упакован **UPX**, распаковка не составляет проблем: набираем ***upx –d*** прямо в IDA 6.3.

Сразу заметно, что программа была создана на основе библиотеки MFC, которая и объясняет большой размер исполняемого файла (1Мб в упакованном виде и 2Мб в распакованном) , а также облегчает ее анализ в IDA:

Рис. 3 Голубой цвет – распознанные IDA функции MFC

Пробежавшись по ссылкам на текстовые строки, найдем адрес функции, отвечающий за проверку ключей:

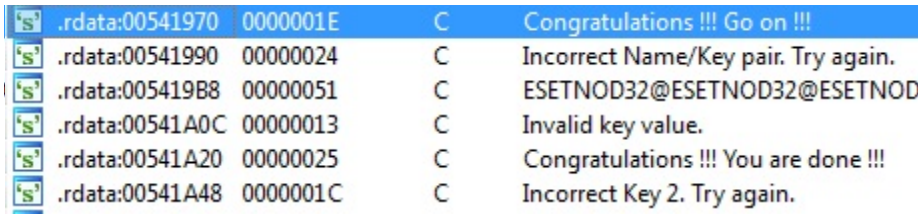


Рис. 4 Ищем функции проверки

Для упрощения анализа мы импортируем метки из IDA в OllyDBG, используя импорт в map-файл, и, загружая его в OllyDBG, с помощью плагина **GoDup** (http://www.openrce.org/downloads/details/103/GoDup). (*я лично пользовался отладчиком IDA, поэтому этот шаг пропустил*).

А теперь приступим к анализу.

3. Первая часть

Быстрый анализ показывает, что ключ – это закодированная base64 строка и длина ключа, по крайней мере, должна быть не менее 0x3F символов. Затем запускается следующий алгоритм-проверка:

Жмем «CHECK STAGE 1»:

00401990: Btn_Check_Stage_1
{
Имя «***Name***» - длина до 256 символов
Ключ «***Key_1***» - длина до 1024 символов
00401AEC: Проверяем длину ключа – должна быть не менее 64 байт (0x3F)
 # Декодируем Base64-строку ключа
00401B11: ***serialDecoded*** = ***Key_1***.decode("base64")

00401B28: EAX = **Check_1**(***Name*** (*ECX), len(***Name***), ***serialDecoded***, len(***serialDecoded***))
{
 # Подготавливаем (обнуляя) массив scrambledTab, размещенный на стеке. Размер массива 80 байт.
00401788: ***scrambledTab[0:76] = 0***

 # Важный момент: отслеживайте использование переменной exitFun. Эта переменная – просто ссылка, указывающая либо на BadBoy-функцию, либо на GoodBoy-функцию
00401794: ***exitFun*** = offset **BadBoy-функция** (адрес **00401740**)

0040179C: If len(***serialDecoded***) <= 64
exitFun = offset **BadBoy-функция**
jmp **show_message**

 # paddingName – Name дополняется или урезается до 80 символов. Заполнитель padding = «ESETNOD32@».
Например, для Name = «xxxxxx».
paddingName = «xxxxxxD32@ESETNOD32@....ESETNOD32@»
004017C6: ***paddingName = Name + padding***
 # Чуть позже рассмотрим функцию hash512. Главное, что мы должны знать, она возвращает 64-байтный хэш.
004017C6: ***H*** = **hash512**(***paddingName***)

 # Первая проверка. Берем первые 64 байта serialDecoded - serialDecoded[:64], делаем реверс отобранных 64 байт - serialDecoded[:64] [::-1], и сравниваем с хэшом H
0040180E: If ***serialDecoded***[:64][::-1] != ***H***
jmp **show_message** (помним ***exitFun*** указывает на **BadBoy-функцию**)

 # Копируем хэш H в блок scrambledTab (размещен на стеке, длина 64 байта)
0040181F: ***scrambledTab*** = ***H*** (или тоже самое, что ***serialDecoded***[:64][::-1])

 # Блок scrambledTab XOR'им с содержимым serialDecoded в цикле поблочно.
 # Если внимательно присмотреться, то этот цикл можно представить как scrambledTab = H XOR H[::-1] XOR egg, или scrambledTab = serialDecoded[:64] [::-1] XOR serialDecoded[:64] XOR egg[0:64], где egg[0:64] – это часть ключа serialDecoded от 64 байт и выше. Условимся, что размер egg кратен 64 байтам.
00401840 – 00401856: for i in len(***serialDecoded***) :
scrambledTab[i%64] ^= ***serialDecoded***[i]

 # Переходим к более интересной части. Далее содержимое scrambledTab «перемешиваем» по алгоритму.
def scramble(t) :
j = 0
for i in xrange(64) :

c = t[i]%80 читаем значение из массива
j = (t[c%80] + j)%80 вычисляем индекс j (возможные значения – 0..79)
t[j], t[i] = t[i], t[j] меняем значения местами t[i] ↔ t[j]

Обратите внимание, сначала мы оперировали только 64-ю байтами scrambledTab, а тут произошло «как бы» расширение массива до 80 байт. Т.е. у нас появилось контролируемое поле в 16 байт: от 64 до 80. И оно нам еще понадобиться.

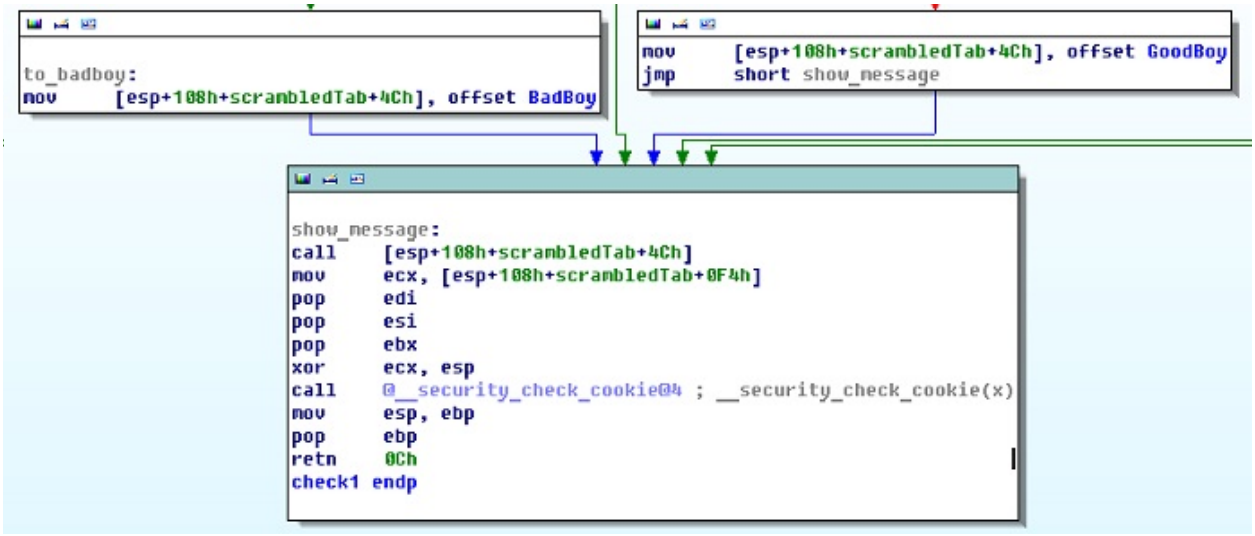
00401860 – 004018FF: scramble(***scrambledTab***)

 # А вот и вторая проверка:
00401905: If ***scrambledTab*****[68 : 72]** != **[0xA, 0xB, 0xC, 0xD]** :
exitFun = offset **BadBoy-функция**
jmp **show_message**
 # И еще одно дополнение. Считаем хэш от scrambledTab и сравниваем первые 12 байт с hardcoded-байтами.
00401944: If **hash512**(***scrambledTab***)[:12] != [0xCA, 0x8A, 0x57, 0x12, 0x78, 0xB6, 0xCA, 0xEF, 0x78, 0x56, 0x34, 0x12] :
jmp **show_message**
Else:
exitFun = offset **GoodBoy-функция**
jmp **show_message**

 # Ого, хэш сравнивается с фиксированным значением. Неужели здесь необходима атака на хэш: поиск коллизий или слабостей в алгоритме хэша. На самом деле все проще. Допустим, это условие НИКОГДА не выполнится. Выполнение пойдет по ветке «jmp show_message; CALL exitFun». Нам нужен контроль над exitFun. exitFun хранится на стеке, а не пересекается ли он с нашим массивом scrambledTab?
0: первые 64 байта scrambledTab
64: DWORD = 0
68: DWORD должно быть 0x0D0C0B0A (смотрите проверку чуть выше)
72: DWORD = 0
76: DWORD exitFun (опа! можно брать под контроль!)

show_message:
00401967 CALL ***exitFun***
}// **00401982**
} //**401BFE**

Подытожим.
Первую часть проверки легко решить, нам нужно просто закодировать в base64 реверс хэша от дополненного имени. Вторая часть, на первый взгляд, кажется, немного сложнее. Мы должны найти строку байт, соответствующее конкретному значению хэш-функции **hash512**, и обратить алгоритм «перемешивания», чтобы вычислить действительный ключ.
После изучения кода хэш-функции, мы обнаружили, что это хэш-функция **Whirlpool** (благодаря таблице замен), и мы не нашли в ее коде никаких «черных ходов». Значит, должно быть что-то другое.
Внимательный читатель мог заметить, что массив ***scrambledTab*** инициализируется 64 байтами (хэшем имени), но далее рассматривается как 80-ти байтный массив в функции «перемешивания». После переопределения ***scrambledTab*** как 80-ти байтный массив в IDA, мы увидели, что ***exitFun*** стал частью ***scrambledTab*** **[76:80]**:



exitFun может указывать либо на **BadBoy-функцию** (**00401740**), либо на **GoodBoy-функцию** (**00401720**). Адреса **BadBoy** и **GoodBoy** функций отличаются только на младший байт (0x40 для **BadBoy-функции**, 0x20 для **GoodBoy**). Так что, используя функцию «перемешивания», мы можем переписать младший байт адреса **BadBoy** на 0x20 и записать проверочные значения ([0xA, 0xB, 0xC, 0xD]) в ***scrambledTab*** [68:72] и мы получим goodboy-сообщение, и первая часть будет пройдена.

Далее автор решения приводит алгоритм умного брутфорса, реализующий запись необходимых значений байт по фиксированным адресам (а также предлагает кейген по созданию «красивых» ключей).

*Я же просто покажу ручной способ нахождения соответствующей **scrambledTab.** Дешево и сердито. При заполнении **scrambledTab** единственная вещь, за которой надо следить, это индекс j. Напомним, алгоритм «перемешивания» **scrambledTab:***

```
def scramble(t) :
  j = 0
  for i in xrange(64) :
    c = t[i]%80 читаем значение из массива
    j = (t[c%80] + j)%80 вычисляем индекс j (возможные значения – 0..79)
    t[j], t[i] = t[i], t[j] меняем значения местами t[i] ↔ t[j]
```

Что мы должны установить? ***scrambledTab*** [68:72] = [0xA, 0xB, 0xC, 0xD], ***scrambledTab*** [76] = [0x20]

*Создаем массив scrambledTab = [0]*80 (80 байт, заполненный нулями).*
i = 0: Пишем в [i] = 0x0A. Тогда c = [0]= 0A, пишем в [c] = [0x0A] =начальное значение j = 0x44 (68).
Держим в уме, что алгоритм выполнив [0] ↔ [68] запишет по смещению 68 нужное нам значение 0x0A.

0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	….	0x20	0x21	0x22	….
0A										0x44									

i = 1: Пишем в [i] = 0x0B. Тогда c = [1]= 0B, пишем в [c] = [0x0B] = приращение j = 0x01 (68+1=69).
Держим в уме, что алгоритм выполнив [1] ↔ [69] запишет по смещению 69 нужное нам значение 0x0B.

0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	….	0x20	0x21	0x22	….
0A	0B									0x44	1								

i = 2: Пишем в [i] = 0x0C. Тогда c = [2]= 0C, пишем в [c] = [0x0B] = приращение j = 0x01 (69+1=70).
Держим в уме, что алгоритм выполнив [2] ↔ [70] запишет по смещению 70 нужное нам значение 0x0C.

0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	….	0x20	0x21	0x22	….
0A	0B	0C								0x44	1	1							

i = 3: Пишем в [i] = 0x0D. Тогда c = [3]= 0D, пишем в [c] = [0x0D] = приращение j = 0x01 (70+1=71).
Держим в уме, что алгоритм выполнив [3] ↔ [71] запишет по смещению 71 нужное нам значение 0x0D.

0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	….	0x20	0x21	0x22	….
0A	0B	0C	0D							0x44	1	1	1						

i = 4: Пишем в [i] = 0x20. Тогда c = [4]= 0x20, пишем в [c] = [0x20] = приращение j = 0x05 (71+5=76). Держим в уме, что алгоритм выполнив [4] ↔ [76] запишет по смещению 76 нужное нам значение 0x20.

0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	….	0x20	0x21	0x22	….
0A	0B	0C	0D	20						0x44	1	1	1			5			

Все необходимые значения мы записали*. А теперь начинается импровизация. Надо задать приращение индекса j такое, чтобы при дальнейшем выполнении алгоритма - j никогда не попадал на фиксированные адреса.*
i = 5: Пишем в [i] = допустим 0x21. Тогда c = [5]= 0x21, пишем в [c] = [0x21] = приращение j = допустим 0x14 (76+20=96 % 80 = 16). Держим в уме, что алгоритм выполнив [5] ↔ [16] запишет по смещению 16 значение 0x21.

0	1	2	3	4	5	6	7	8	9	0A	0B	0C	0D	0E	….	0x20	0x21	0x22	….
0A	0B	0C	0D	0x20	0x21					0x44	1	1	1			5	0x14		

И т.д.

Итак, начальное значение массива ***scrambledTab*** *(первые 64 байта****)*** *перед выполнением функции «перемешивания»****:****.*
[0x0A, 0x0B , 0x0C, 0x0D, 0x20, 0x21, 0, 0, 0, 0, 0x44, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0x14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Зная, начальное заполнение ***scrambledTab*** *(первые 64 байта****)*** *можно полностью восстановить ключ.*
scrambledTab = serialDecoded[:64] [::-1] XOR serialDecoded[:64] XOR egg[0:64]
egg[0:64] = serialDecoded[:64] [::-1] XOR serialDecoded[:64] XOR scrambledTab
Тогда ключ равен:
serialbinary = serialDecoded[:64] [::-1] + egg[0:64]
Кодируем в base64:
serial = base64.b64encode(serialbinary)

Кейген stage1.py

```
# -*- coding: cp1251 -*-
import base64
try :
  from whirlpool import Whirlpool
except :
  print "Для работы этого кейгена необходима реализация хэша Whirpool, качать отсюда:\
    http://www.bjrn.se/code/whirlpoolpy.txt"

# Укажите имя
name = "yourname"
# Дополненное имя
namepad = name + ("ESETNOD32@"*8)[len(name):]
# Считаем хэш от имени
hash_name = [ord(c) for c in Whirlpool(namepad).digest()]

scrambledTab = [0x0A, 0x0B , 0x0C, 0x0D, 0x20, 0x21, 0, 0, 0, 0, 0x44, 1, 1, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 0x14, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

serialbinary = [0]*64
# for i in xrange(64):
#    serialbinary [i] = hash_name[i] ^ hash_name[::-1][i] ^ scrambledTab[i]
# Краткая форма записи цикла
serialbinary = [a^b^c for a, b, c in zip(hash_name, hash_name[::-1], scrambledTab)]

serial = "".join(chr(i) for i in hash_name[::-1])
serial += "".join(chr(i) for i in serialbinary)
```

```
# Кодируем в base64
serial = base64.b64encode(serial)
print ("[+]Name: "+name)
print ("[+]Key 1: "+serial)
```

4. Вторая часть

Функция проверки второго ключа стартует с адреса **00401C10**. Ключ должен состоять из 12 шестнадцатеричных символов. Эти символы используются для преобразования 64 битного числа ***qword*** по следующему алгоритму:

Жмем «CHECK STAGE 2»:

00401C10: Btn_Check_Stage_2
{
    # Стартовое значение qword (константа)
    00401C52: ***qword*** = **0x7A83B471B11B7780** (в десятичном виде 8828098094971975552)
    # Переводим символы ключа в верхний регистр
    00401CA9: ***serial*** = Upper(***serial***)
    # Первый этап - трансформация qword по нашему ключу
    00401D26 – 00401D76:
        for c in serial :
            # Каждый символ ключа – это hex-цифра (0..9A..F)
            i = int(c,16)
            word = **to_16b**(***qword***) //**00402290**
            if ***word*** & (1 << i) :
                word &= ~(1 << i)
            else :
                word |= (1 << i)
            qword = **mutate**(***qword***, ***word***) //**004023C0**
            qword = **shift**(***qword***) //**00402210**

    # Второй этап – «считаем» вхождения элементов таблицы qwordTable[64][8] (адрес 00560128) в значение qword
    00401D7D: EAX = **check**(***qword***)
    {
        r = 0
        for i in xrange(64) :
            nb_same_set_bits = 0
            for j in xrange(8) :
                nb_same_set_bits += **has_same_set_bits**(***qword***, ***qwordTable***[i][j])
            if nb_same_set_bits & 1 :
                r |= 1 << i
    if r == **0x55BDEC8E23A0EF32** :
            return 1   // "Good Boy"
        else :
            return 0   // "Bad Boy"
    } //**00401D7D**
    # Показываем сообщение "Good Boy" или "Bad Boy"
} //**00401C10**

Рассмотрим первый этап. Возможно, здесь есть что упростить.

i = int(c, 16)
word = **to_16b**(***qword***)
if ***word*** & (1 << i) :
  word &= ~(1 << i)
else :
  word |= (1 << i)
qword = **mutate**(***qword***, ***word***)
	def **to_16b**(***qword***) :
	  idx = 1
	  word = 0
	  for i in xrange(16) :
	  if qword & (1 << idx) :
	  word |= 1 << i
	  idx = (idx + 29) % 64
	  return word
	def **mutate**(***qword***, ***word***) :
		Цикл
		idx = 1
		for i in xrange(16):
		  idx = (idx + 29) % 64
		можно представить как массив-константу
		int_to_idx = [1, 30, 59, 24, 53, 18, 47, 12, 41, 6, 35, 0, 29, 58, 23, 52]
		
		Обратите внимание на битовые операции.

	idx = 1 for i in xrange(16) : if word & (1 << i) : qword |= 1 << idx else : qword &= ~(1 << idx) idx = (idx + 29) % 64 return qword	В частности в функции **mutate** и между вызовами **to_16b** и **mutate**. Ничего не напоминает. Вопрос: как можно выразить операцию XOR через AND (&), OR (|), NOT (~)? Ответ: a XOR b или (a | b) & (~a | ~b) = (a & ~a) | (b & ~a) | (a | ~b) | (b & ~b) = **(b & ~a) | (a | ~b)**
qword = **shift**(***qword***)	def **shift**(***qword***) : qword_masked = qword & **0x621AC745FB723ED1** nb_bits_set = 0 for i in xrange(64) : if qword_masked & (1 << i) : nb_bits_set += 1 return ((qword << 1) | (nb_bits_set & 1)) & ((1 << 64) - 1)	Накладываем на текущее значение ***qword*** **маску-константу** и считаем количество установленных бит (те, что стоят в 1). В случае если количество нечетно, то функция возвращает ((***qword*** << 1) | 1)), иначе (***qword*** << 1) bit = 0 for i in xrange(64) : if **0x621AC745FB723ED1** & (1 << i) : bit = ((qword << i) & 1) ^ bit return ((qword << 1) | bit))

Первый этап – трансформация qword (после упрощения)

```
def shift(qword) :
    bit = 0
    for i in xrange(64) :
      if 0x621AC745FB723ED1 & (1 << i) :
        bit = ((qword << i) & 1) ^ bit
   return ((qword << 1) | bit) & 0xFFFFFFFFFFFFFFFF

serial = "012345678AB"
qword = 0x7A83B471B11B7780
int_to_idx = [1, 30, 59, 24, 53, 18, 47, 12, 41, 6, 35, 0, 29, 58, 23, 52]
for c in serial:
   qword ^= 1 << int_to_idx[int(c,16)]
   qword = shift(qword)

 # Как покажет практика, при трансформации на входе наши ключи (поле «Key 2») могут быть разными, но при окончании первого этапа значение qword примет некое фиксированное значение.
print(hex(qword))   # 0x19473B90FD7C0372
```

На втором этапе нам нужна сама таблица qwordTable[64][8]. Небольшой скрипт в IDA дампит таблицу в окно вывода для сохранения в файл.

```
        auto a, i, cnt, count;
        a = 0x560128;
        count = 0x1000;
        cnt = 0;
        for (i = 0; i < count; i = i+8) {
                if ((cnt % 8) == 0 ) Message("\n");
                Message("0x%08x%08x, ", Dword(a+i+4),Dword(a+i));
                cnt = cnt+1;
        }

qwordTable  =
[
[0x0000000000000004, 0x0000000000000091, 0x00000000000000f9, 0x0000000000000048,
0x0000000000000013, 0x00000000000000c3, 0x000000000000004a, 0x0000000000000089],
…
[0x4000000000000000, 0x1097c1a8ace792f7, 0x005778ad02765668, 0x009b685219b53ba6,
0x09652bf15909839e, 0x10b409f5185e0cc5, 0x0e644b8167b5d108, 0x01cdfea6425690e7,]
]
```

Второй этап:
 # has_same_set_bits – функция ищет вхождения «b» в «a». Чтобы было понятней, приведу простой пример. Пусть у нас есть число «a» равное 14 = 0x0E = бинарный вид 1110. И есть массив чисел «b»: 8, 4, 1, 16. Тогда ищем вхождения «b» в «a».
8 входит в 14? Да. 8 = бинарный вид 1000 = «входит» в 1110. Функция вернет 1.
4 входит в 14? Да. 4 = бинарный вид 0100 = «входит» в 1110. Функция вернет 1.
1 входит в 14? Нет. 1 = бинарный вид 0001 = «не входит» в 1110. Функция вернет 0.
16 входит в 14? Да. 16 = бинарный вид 10000 = «не входит» в 1110. Функция вернет 0.
В нашем случае «a» - это qword , «b» - это массив qwordTable[64][8]

```
def has_same_set_bits(a, b) :
    for i in xrange(64) :
        if b & (1 << i) and not (a & 1 << i) :
            return 0
    return 1

r = 0
for i in xrange(64) :
    nb_same_set_bits = 0
    for j in xrange(8) :
        nb_same_set_bits += has_same_set_bits(qword, qwordTable[i][j])
    if nb_same_set_bits & 1 :
        r |= 1 << i

if r == 0x55BDEC8E23A0EF32 :
    return 1    // "Good Boy"
else :
    return 0    // "Bad Boy"
```

Может и существует некое элегантное решение второй части **crack-me**, но мы его не нашли. И т.к. мы хотели решить задачку как можно быстрее, то решили использовать инструмент **Microsoft Theorem Prover: Z3 (http://research.microsoft.com/en-us/um/redmond/projects/z3/ - это мощный производительный решатель теорем, разработанный подразделением Microsoft Research)**. Инструмент дружественен с Python API, принят сообществом, бесплатен и прост в использовании. Он является идеальным инструментом для решения этой задачки.
Все, что нам нужно сделать, это перевести различные операции над ключом в уравнения Z3 и пусть Z3 найдет для нас ответ.
Целые числа представим в виде битовых векторов, различные возможные варианты значений переводим в формат OR-уравнений.

*Чтобы последующий код на Python был более понятным, разложим алгоритм второй части на выражения в формате Z3. k0, k1, .. k11 – это символы нашего ключа (то, что мы ищем. Возможные значения каждого 0..9A..F, длина ключа – 12 символов, поэтому переменных всего 12: от k0 до k11). x_0 .. x_11- возможные текущие преобразования **qword** по соответствующему символу ключа k0 .. k11.*

Первый этап – трансформация qword

```
qword = 0x7A83B471B11B7780
int_to_idx = [1, 30, 59, 24, 53, 18, 47, 12, 41, 6, 35, 0, 29, 58, 23, 52]
FALSE OR

  (k0 == 0 & x_0 == shift(qword ^ (1 << int_to_idx[0])))
OR
  (k0 == 1 & x_0 == shift(qword ^ (1 << int_to_idx[1])))
OR
........
OR
  (k0 == 11 & x_0 == shift(qword ^ (1 << int_to_idx[11])))
                                                          OR
  (k1 == 0 & x_1 == shift(x_0 ^ (1 << int_to_idx[0])))
OR
  (k1 == 1 & x_1 == shift(x_0 ^ (1 << int_to_idx[1])))
OR
........
OR
  (k1 == 11 & x_1 == shift(x_0 ^ (1 << int_to_idx[11])))
                                                          OR
  (k1 == 0 & x_11 == shift(x_10 ^ (1 << int_to_idx[0])))
OR
  (k1 == 1 & x_11 == shift(x_10 ^ (1 << int_to_idx[1])))
OR
........
OR
  (k1 == 11 & x_11 == shift(x_10 ^ (1 << int_to_idx[11])))
```

Пусть выражение вида
xxx ^ (1 << int_to_idx[n]))
обозначим **bv**
Тогда функцию **shift** раскладываем на битовые операции:
If **0x621AC745FB723ED1 & (1 << m_j):**
0 ^ (LShR(bv,m_j) & 1) ^ …
Для всех m от 0 до 64

Второй этап – задаем условия

*Результатом первого этапа является **x_11**. Теперь для него надо задать условие. Для каждого v ← **qwordTable**[i][0-7] (где i от 0 до 64) считаем вхождения в **x_11**.*

bit = 0
for v ← qwordTable[i][j ← 0-7]
for k от 0 до 64
if **v & (1 << m_k):**
1 & (LShR(x_11, m_k) & 1) & …
bit = bit ^ (1 & (LShR(x_11, m_{ik}) & 1) & …)

Далее, проверка с константой **0x55BDEC8E23A0EF32**. Проверяем бит за битом.

If (**0x55BDEC8E23A0EF32** & (1 << i)):
Добавляем в Z3 условие:
0 ^ (1 & (LShR(x_11, m_{ik}) & 1) & …) ^ (1 & (LShR(x_11, m_{ik}) & 1) & …) ^ … == 1
Else
Добавляем в Z3 условие:
0 ^ (1 & (LShR(x_11, m_{ik}) & 1) & …) ^ (1 & (LShR(x_11, m_{ik}) & 1) & …) ^ … == 0

Следующий код находит ключ менее чем за 2 минуты:

```
qwordTable = [  [..], [..], … [..]  ]
qword = 0x7A83B471B11B7780
check_v = 0x55BDEC8E23A0EF32
shift_v = 0x621AC745FB723ED1

bitvectors = [BitVecVal(qword, 64)]

# Создаем объект "решатель" Z3
solver = Solver()

for i in xrange(12) :
  # Создаем битовые вектора x_0 .. x_11 размером 64 бита
  bitvectors.append(BitVec('x_%d'%i, 64))

key = []
for i in xrange(12) :
  # Значения этих переменных мы должны найти k0 .. k11
  key.append(Int("k%d"%i))

# Первый этап –  трансформация qword
for i in xrange(1, 13) :
  eq = False
  for j, k in enumerate([1, 30, 59, 24, 53, 18, 47, 12, 41, 6, 35, 0, 29, 58, 23, 52]) :
    sub_eq = (key[i-1] == j)
    bv = (bitvectors[i-1] ^ BitVecVal(1 << k, 64))
    bit = 0
    for l in xrange(64) :
      if shift_v & (1 << l) :
        bit = (LShR(bv, l) & 1) ^ bit
    sub_eq = And(sub_eq, bitvectors[i] == ((bv << 1) | bit))
    eq = Or(eq, sub_eq)
  solver.append(eq)

# Второй этап – задаем условия
for i, t in enumerate(qwordTable) :
  bit = 0
  for v in t :
    has_same_set_bits = 1
    for j in range(64) :
      if v & (1 << j) :
        has_same_set_bits = has_same_set_bits & (LShR(bitvectors[12], j) & 1)
    bit ^= has_same_set_bits
  if check_v & (1 << i) :
    solver.add(bit == 1)
  else :
    solver.add(bit == 0)

# Получаем результат
solver.check()
solution = solver.model()

# Собираем ключ из k0 .. k11
r = ""
for i in xrange(12) :
  r += "%X"%int(str(solution[key[i]]))

# На печать
print ("[+]Key 2: " + r)
```

Приведенный код можно еще более оптимизировать, заменив битовые вектора булевыми переменными. Тем самым время выполнения скрипта сокращается до 30 секунд вместо 100 изначальных.

Этот код я приводить не буду, желающие могут посмотреть в первоисточнике – в блоге ***http://blog-oppida.blogspot.fr/2012/08/solution-for-eset-blackhat-us-challenge.html***.

Примеры решений для crack-me:

[+]Name: solutionmes
[+]Key 1:
BvD9A1JC2Gd2TSinVSdU3tC7P8b4o39BSYafrP4uGYtrwe92cT+UVrtieABoSHY9JhfJ0XTY6hvSkti2A7IGqKT
9Qw3Eu0q1bae00oXvQ/jtzXeu+Nsd+h8SoN2Iwdjg5czBiN2gEh/6Hdv4rnfN7fhD7oTT8KdttUqa5ABP9q4=

[+]Name: BlackHat@2012
[+]Key 1:
FKT3FbegEcbkFx8ybuGRQJnLSqWj/
u1C9bAl93Wx8SDgiQwyMNg1BhBdpLXRjnhx5piD9V6V5MZRZmJ65HfLNihkjPzt43eXIvPObZpjCabos8R0Flq
wUvOF/cdHvXjAxWy9R8f9hfNSsFoWdMSz6KYJYptsivMil3fCzfGAbyI=

[+]Key 2: 93A52ECFAC23
[+]Key 2: B0CEDE7F8323
[+]Key 2: 94A0353FA32B
[+]Key 2: 2BA52E7FAC23
[+]Key 2: 966132348323