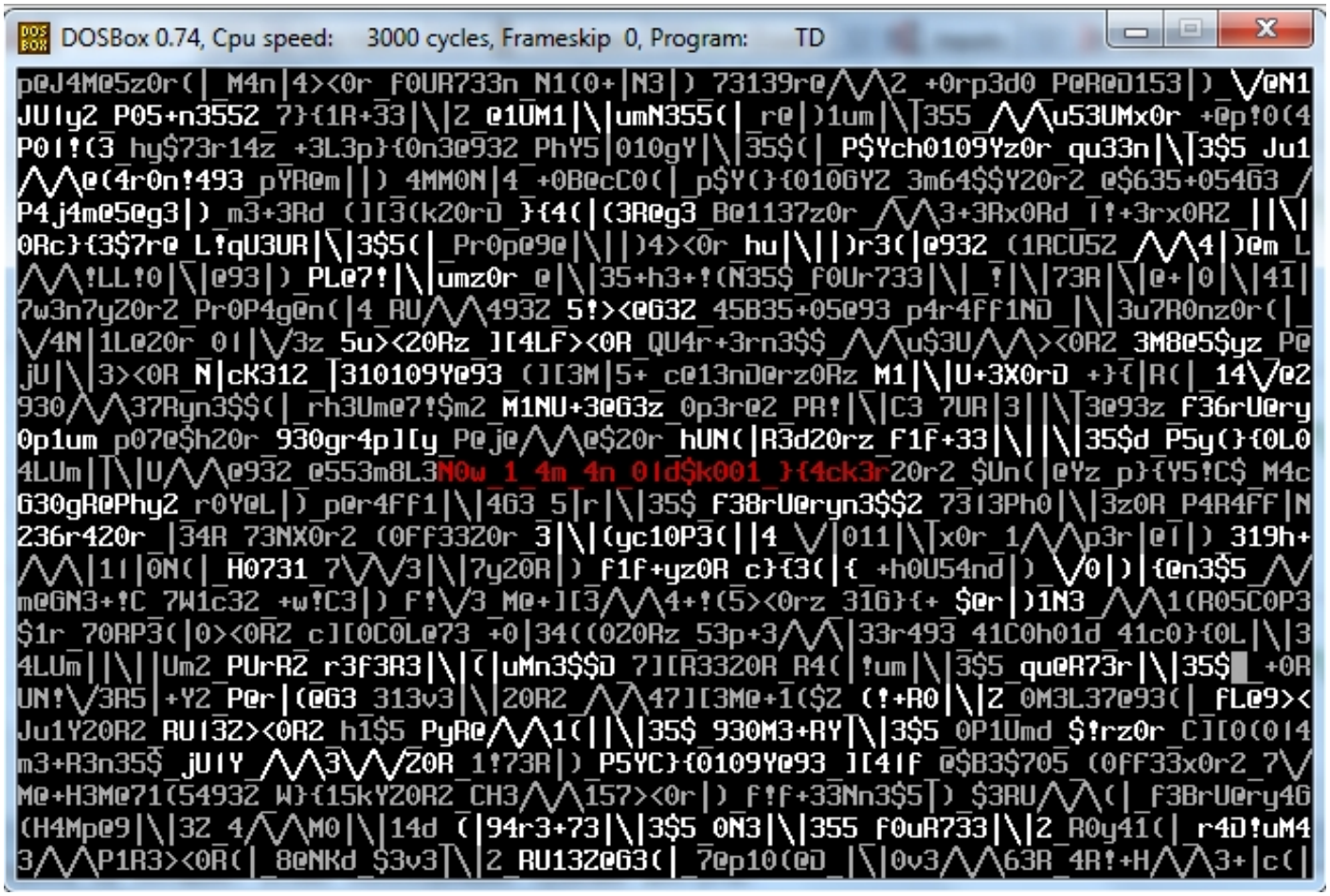
Решение заданий RuCTF2013 Quals от WildRide

reverse100

Программа для DOS. Если открыть её в текстовом редакторе, можно увидеть строчку "PKLITE Copr. 1990-92" , свидетельствующую о том, что программа запакована. Распаковывал с помощью утилиты UNP 4.11. В итоге получаем исходный файл размером 106 КБ. Программа не содержит никаких защитных механизмов. Поэтому спокойно её дизассемблируем. Практически в самом начале располагается основной цикл считывания нажатий клавиш. Помимо управляющих клавиш ожидается ввод в правильном порядке клавиш с заданными скан-кодами (0x19, 0x12, 0x31, 0x2E, 0x17, 0x26), которым соответствует слово PENCIL. Запускаем программу под DOSBox'ом, вводим слово - получаем ключ:



crypto100

Дан зашифрованный файл и ключ. Файл начинается со строчки

```
"Salted__",
```

которая является стандартной сигнатурой для файлов, зашифрованных openssl с солью. Остаётся только подобрать алгоритм шифрования. Как выяснилось позже использовался алгоритм RC5, который по умолчанию не включен в готовые сборки. Поэтому необходимо собрать openssl с поддержкой RC5 из исходников:
`./configure enable-rc5`
Для перебора всех алгоритмов использовался скрипт:

Listing 1: brut.pl

```
@buf = qw (
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
aes-256-cbc       aes-256-ecb       base64            bf
```

```
bf-cbc              bf-cfb              bf-ecb              bf-ofb
camellia-128-cbc    camellia-128-ecb    camellia-192-cbc    camellia-192-ecb
camellia-256-cbc    camellia-256-ecb    cast                cast-cbc
cast5-cbc           cast5-cfb           cast5-ecb           cast5-ofb
des                 des-cbc             des-cfb             des-ecb
des-ede             des-ede-cbc         des-ede-cfb         des-ede-ofb
des-ede3            des-ede3-cbc        des-ede3-cfb        des-ede3-ofb
des-ofb             des3                desx                idea
idea-cbc            idea-cfb            idea-ecb            idea-ofb
rc2                 rc2-40-cbc          rc2-64-cbc          rc2-cbc
rc2-cfb             rc2-ecb             rc2-ofb             rc4
rc4-40              rc5                 rc5-cbc             rc5-cfb
rc5-ecb             rc5-ofb             seed                seed-cbc
seed-cfb            seed-ecb            seed-ofb
);

for ($i=0;$i<=$#buf;$i++) {
  printf "$buf[$i]:\n";
  system("openssl.exe enc -d -pass file:password.txt -in ct.enc -$buf[$i]");
  print "\n";
  }
```

crypto200

Рюкзачная криптосистема. Доступны публичные ключи, секретный модуль, секретный множитель и шифртекст. Таким образом, необходимо просто написать функцию расшифровки.

Listing 2: dec.pl

```
def xgcd(x,y):
    a0=1; b0=0
    a1=0; b1=1
    if x<0:
      x = -x
      a0 = -1
    if y<0:
      y = -y
      b1 = -1
    if x<y:
      x,y,a0,b0,a1,b1 = y,x,a1,b1,a0,b0
    while 1:
      times = x // y
      x -= times*y
      a0 -= times*a1
      b0 -= times*b1
      if x==0:
        break
      x,y,a0,b0,a1,b1 = y,x,a1,b1,a0,b0
    return [y,a1,b1]

n = 19928531897866896652755134251299725081663770927474925998329207769944036 9
t = 32416190071
n1 = 73510345939
n2 = 123577956023227266490471
n3 = 24374926927665135545271756945647199 89
fn = (n1 - 1) * (n2 - 1) * (n3 - 1)

u = xgcd (t, n)[1]

def generate_private_key(public_key):
  global n
```

```
  global u
  return list(map(lambda x: (u * x) % n, public_key))


def generate_public_key(private_key):
  global n
  global t
  return list(map(lambda x: (t * x) % n, private_key))

def crypt(open_text, public_key):
  bts = []
  [bts.extend([int(b) for b in '00000000'[len(bin(ord(c))[2:]):] + bin(ord(c))[2:]]) for c
     in open_text]
  return [sum(map(lambda x: x[0] * x[1] ,zip(blk, public_key))) for blk in [bts[i * 128:(i
     +1) * 128] for i in range(len(open_text) // 16)]]

public_key = [...]

cipher_text = [...]


def decrypt(cipher_text, private_key):
  cipher_text = (cipher_text * u) % n
  cipher_sum = []
  bts = []
  sort_private_key = sorted(private_key, reverse=True)
  for i in sort_private_key:
    if (cipher_text >= i):
      cipher_sum.append(i)
      cipher_text -= i
  for i in private_key:
    if (i in cipher_sum):
      bts.append(1)
    else:
      bts.append(0)

  return bts

private_key = generate_private_key (public_key)
plain_bits = []
for text in cipher_text:
  plain_bits += decrypt (text, private_key)

plain_text = []
for i in range (len(plain_bits) // 8):
  block = plain_bits[(i)*8:(i+1)*8]
  byte = 0
  for i in range(7, -1 ,-1):
    byte += block[i] * (2**(7-i))
  plain_text.append (chr(byte))
print (''.join(plain_text))
```

crypto300

Шифрование осуществляется следующим образом: блоки открытого текста по 16 байт объединяются в одной большое число, шифртекстом блока являются 10 двухбайтовых остатков от деления на 10 двухбайтовых ключей (ключи попарно взаимнопросты). Следовательно, для расшифрования можно воспользоваться китайской теоремой об остатках и по 10 остаткам восстановить число, дающее эти остатки по заданным модулям. Таких чисел будет бесконечно много, они определяются с точностью до слогаемого, кратного произведению всех ключей (т.е. лежат в одном классе вычетов по этому модулю). Предполагаем, что необходимо брать наименьшее число

(при расшифровке это подтвердилось). Для расшифровки необходимо найти ключи. Можем воспользоваться информацией о формате файла. Первые 16 байт формата PDF можно попробовать перебрать (некоторые поля неизменны, некоторые изменяются в известных пределах). Для каждой возможной сигнатуры перебором по отдельности находим возможные ключи. После чего на них пытаемся расшифровать файл.

Listing 3: brut.py

```
#!/usr/bin/env python3
from struct import pack, unpack
import sys


def xgcd(x,y):
    a0=1; b0=0
    a1=0; b1=1
    if x<0:
      x *= -1
      a0 = -1
    if y<0:
      y *= -1
      b1 = -1
    if x<y:
      x,y,a0,b0,a1,b1 = y,x,a1,b1,a0,b0
    while 1:
      times = x // y
      x -= times*y
      a0 -= times*a1
      b0 -= times*b1
      if x==0:
        break
      x,y,a0,b0,a1,b1 = y,x,a1,b1,a0,b0
    return [y,a1,b1]


def invmod(x,p):
  [gcd,a,b] = xgcd(x,p)
  if gcd != 1:
    return 0
  if a<0:
    a += p;
  return a

brut_count = 0

def decrypt(enc, key):
  m = 1
  for k in key:
    m *= k

  global brut_count

  mi = list (map (lambda x: m // x, key) )
  inv = list (map (lambda x: invmod (m // x, x), key) )

  block = enc[0 : 2 * KEY_LEN]
  block = list(unpack('H' * KEY_LEN, block))

  d = 0
  for j in range(0,KEY_LEN):
    d += mi[j] * inv[j] * block[j]
  d %= m
```

```
  dec = []
  for j in range(BLOCK_SIZE-1, -1, -1):
    dec.append ((d // (256**j)) & 0xFF)

  if (dec[0] != 0x25) or (dec[1] != 0x50) or (dec[2] != 0x44) or (dec[3] != 0x46):
    print (brut_count, 'error', dec)
    return
  else:
    print (brut_count, 'find', len(dec), dec)
    dec_file = open('out' + str(brut_count), 'wb')
    brut_count += 1

  for i in range((len(enc) // 2*KEY_LEN) - 2):
    block = enc[2 * KEY_LEN * i : 2 * KEY_LEN * (i + 1)]
    if len(block) != 2 * KEY_LEN:
      return
    else:
      block = list(unpack('H' * KEY_LEN, block))
    d = 0
    for j in range(0,KEY_LEN):
      d += mi[j] * inv[j] * block[j]
    d %= m
    dec = []
    for j in range(BLOCK_SIZE-1, -1, -1):
      dec.append ((d // (256**j)) & 0xFF)

    for d in dec:
      dec_file.write (pack ('B',  d ))
  dec_file.close();

def brut(data_block, enc_block):
  d = 0
  for byte in data_block:
    d *= 256
    d += byte
  count = 0
  l = []
  key = []
  for b in enc_block:
    prekey = []
    for k in range(1, 256**2):
      if d % k == b:
        l.append([k,b])
        prekey.append(k);
        count += 1
    if len(prekey) == 0:
      break;
    else:
      if len(key) == 0:
        for x in prekey:
          key.append ([x])
      else:
        tmpkey = []
        for k in key:
          for x in prekey:
            tmpkey.append (k+[x])
        key = tmpkey;
  print (len(key))
  print ('key', key)
  print ('l', l)
  if count >= 10:
```

```
    for k in key:
      encoded_file = open(ENCODED_FILE, 'rb')
      enc = encoded_file.read()
      encoded_file.close();
      decrypt (enc, k)


KEY_LEN = 10
ENCODED_FILE = "encoded" if len(sys.argv) < 2 else sys.argv[1]
BLOCK_SIZE = 16

key = []

encoded_file = open(ENCODED_FILE, 'rb')
enc = encoded_file.read()
encoded_file.close();
enc_block = enc[: 2*KEY_LEN]
enc_block = list(unpack('H' * KEY_LEN, enc_block))


data_file = open('dh1024.pem', 'rb')
data_block = data_file.read()
data_file.close()
data_block = data_block[:BLOCK_SIZE];

data_block1 = [0x25, 0x50, 0x44, 0x46, 0x2d, 0x31, 0x2e, 0x35, 0x0d, 0x25, 0xe2, 0xe3, 0xcf,
    0xd3, 0x0d, 0x0a]
data_block2 = [0x25, 0x50, 0x44, 0x46, 0x2d, 0x31, 0x2e, 0x35, 0x0a, 0x25, 0xe2, 0xe3, 0xcf,
    0xd3, 0x0a, 0x30]
data_block3 = [0x25, 0x50, 0x44, 0x46, 0x2d, 0x31, 0x2e, 0x35, 0x0d, 0x25, 0xe2, 0xe3, 0xcf,
    0xd3, 0x0d, 0x30]
data_block4 = [0x25, 0x50, 0x44, 0x46, 0x2D, 0x31, 0x2E, 0x35, 0x0D, 0x0A, 0x25, 0xE2, 0xE3,
    0xCF, 0xD3, 0x0D]
data_block5 = [0x25, 0x50, 0x44, 0x46, 0x2d, 0x31, 0x2e, 0x35, 0x0a, 0x25, 0xd0, 0xd4, 0xc5,
    0xd8, 0x0a, 0x30]
data_block6 = [0x25, 0x50, 0x44, 0x46, 0x2d, 0x31, 0x2e, 0x35, 0x0a, 0x25, 0xc7, 0xec, 0x8f,
    0xa2, 0x0a, 0x30]
data_block7 = [0x25, 0x50, 0x44, 0x46, 0x2d, 0x31, 0x2e, 0x35, 0x0a, 0x25, 0xe4, 0xf0, 0xed,
    0xf8, 0x0a, 0x30]

#for i in range(8):
#  data_block1[7] = 0x30 + i
#  brut (data_block1, enc_block)

#for i in range(8):
#  data_block2[7] = 0x30 + i
#  for j in range(10):
#    data_block2[15] = 0x30 + j
#    brut (data_block2, enc_block)

#for i in range(8):
#  data_block3[7] = 0x30 + i
#  for j in range(10):
#    data_block3[15] = 0x30 + j
#    brut (data_block3, enc_block)

# for i in range(8):
 # data_block4[7] = 0x30 + i
 # brut (data_block4, enc_block)

# for i in range(8):
```

```
 # data_block5[7] = 0x30 + i
 # for j in range(10):
   # data_block5[15] = 0x30 + j
   # brut (data_block5, enc_block)

# for i in range(8):
 # data_block6[7] = 0x30 + i
 # for j in range(10):
   # data_block6[15] = 0x30 + j
   # print (i,j)
   # brut (data_block6, enc_block)

# for i in range(8):
 # data_block7[7] = 0x30 + i
 # for j in range(10):
   # data_block7[15] = 0x30 + j
   # brut (data_block7, enc_block)

data_block  = [0x25, 0x50, 0x44, 0x46, 0x2D, 0x31, 0x2E, 0x34, 0x0A, 0x25, 0xC7, 0xEC, 0x8F,
    0xA2, 0x0A, 0x37]
brut (data_block, enc_block)
```

Находим ключ:
`EF B9 C3 D6 CD 49 90 6E EB C1 4F 8A 9F 69 E3 B3 51 B3 1F 30`

В итоге получаем расшифрованный файл - фрагмент книги "Foundations of Cryptography". Долго ищем в нем флаг и ничего не находим. Как оказалось, во время игры файл перезалили. Новая версия файла отличается от предыдущей ровно на одну страницу с флагом.

vuln300

Подсоединяемся к порту:
`nc aloneinthedark.quals.ructf.org 3255`
Из подсказки:
`Mighty Bash is restricted and blinded`
понимаем, что вроде бы запускается bash, но урезанный и слепой. Пробуем что-нибудь вводить - на выходе ничего не получаем. Понимаем, что если это обычная оболочка, запущенная на порту, значит стандартный ввод у неё перенаправлен в сокет, а стандартный вывод и ошибка - куда-то в /dev/null. Значит нужно перенаправить их тоже в сокет. Проверяем:
`echo hello 2>&0 1>&0`
И видим вывод команды. Пробуем запускать другие команды и выясняем, что bash запущен в ограниченном режиме и кроме него ничего больше нет. Значит будем обходится только встроенными командами оболочки. Каждую команду можно сопровождать перенаправлением вывода и ошибки. А можно сделать работу чуть более удобной:
`while [ true ]; do read a; eval $a; done 1>&0 2>&0`
И после этого вводим команды обычным образом:

```
pwd
/
echo *
FLAG_NOGUESSING bin lib64 lost+found
read b < FLAG_NOGUESSING
set
BASH=/bin/bash
...
_=a
a=set
b=7c2b867e94f75bcb9a8e9cdf67e7a334
previous=N
```

```
runlevel=3
```

random500

Задание, на которое была потрачена почти половина времени во время игры, но доделать удалось только после. По постановке задачи передельно простое: дан дамп физической памяти операционной системы Windows 7 SP1 x64 и раздел с зашифрованным с помощью EFS файлом; необходимо прочитать файл. Если файл был открыт во время снятия дампа - его можно попробовать достать из памяти программы просмотра либо из системного кеша. Но первое слишком просто для задания на 500. Второе наоборот достаточно сложно: кусочки файла (вроде бы по 128КБ) будут лежать по отдельности. Готового инструмента не нашел, а писать самостоятельно был не готов. В любом случае сигнатурный поиск графических файлов не дал результатов (хотя некоторые картинки восстановить удалось). Поэтому оставалось только расшифровывать файл на диске. Понятно, что ключи для расшифровки необходимо достать из дампа. Первоначальная мысль, была, что пользователь скорее всего залогинился в системе и, наверно, где-то в памяти должен быть его закрытый ключ. Уверенности добавляло большое количество вхождений сигнатуры открытого ключа в дампе. В процессе работы познакомился с прекрасными утилитами для анализа дампов памяти: Volatility Framework и MoonSols Windows Memory Toolkit. Как извлечь закрытый ключ из дампа придумать так и не удалось. Но зато с помощью Volatility Framework удалось получить значительную информацию о системе. В частности, об открытых файлах. Выяснилось, что зашифрованный файл был открыт. Значит, в памяти закешировался FEK. По умолчанию используется AES256. Для поиска в памяти ключей AES можно использовать "Elcomsoft Forensic Disk Decryptor"либо aeskeyfind (есть в репозитариях Debian'a). Обе они нашли одинаковые ключи, в точ числе один ключе AES256. Осталось только проверить, что это нужный ключ. В пакете linux-ntfs есть модуль ntfsdecrypt (по умолчанию не включен в сборку), расшифровывающий файлы с помощью закрытого ключа, импортированного стандартными средствами Windows. Для экспериментов, модифицировал программу, чтобы она выводила FEK, а на тестовой системе создал шифрованный файл, снял дамп памяти и нашёл в памяти ключи. Ключ, выводимый ntfsdecrypt, и ключ, найденный aeskeyfind, совпали. Дальше оставалось только модифицировать ntfsdecrypt, чтобы она брала уже расшифрованный FEK из файла, а не расшифровывала его из файлового потока.