TrendMicro CTF 2018 Qualifier Writeup by r3kapig

Atum / 2018-09-16 21:28:21 / 浏览数 4444 安全技术 CTF 顶(0) 踩(0)

我们是由Eur3kA和flappypig组成的联合战队r3kapig。本周末，我们参与了趋势科技举办的TrendMicro CTF 2018 Qualifier
并以第十名的成绩成功晋级12月在日本东京举办的TrendMicro CTF 2018 Final。我们决定把我们做出来的题目的writeup发出来分享给大家。
另外我们战队目前正在招募队员，欢迎想与我们一起玩的同学加入我们，尤其是Misc/Crypto的大佬，有意向的同学请联系lgcpku@gmail.com。给大佬们递茶。
由于是国际比赛，所以我们的首发wp为英文版，中文版正在路上～

Analysis-Offense

200

I just modified my callgrind solver to solve this challenge.

```
$ cat oracle.py
#!/usr/bin/python -u
#-*- coding:utf-8 -*-

# Let's exploit easy and quick!
# 1) apt install valgrind
# 2) use callgrind to find instruction count

flag = 'TMCTF{'
n = 0

import os
import sys

# format given by admin
charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789{}"
while True:
   n += 1
   total_call_count = {}
   for i in charset:
       cmd = "valgrind --tool=callgrind --dump-instr=yes --callgrind-out-file=temp/call_count ./oracle '" + flag + i + "A'  2>
       # print(cmd)
       res =  os.popen(cmd).read()

       call_count = res.split("Collected : ")[1].split()[0]
       call_count = int(call_count)
       # total_call_count { 'call_count': [occured_count, occured_by], ... }
       if not total_call_count.get(call_count):
               total_call_count[call_count] = [1, [i]]
       else:
               total_call_count[call_count][0] += 1
               total_call_count[call_count][1].append(i)
       print(n, i, call_count)
   ## get lowest/highest idx,
   idx_call_count = total_call_count.keys()
   print(idx_call_count)
   idx_call_count.sort()
   highest_count_idx = idx_call_count[-1]
   lowest_count_idx = idx_call_count[0]
   # get highest idx
   flag_char = total_call_count[highest_count_idx][1][0]
   flag += flag_char
   print(n, total_call_count, highest_count_idx, flag)
```

300

We get 3 rsa public keys here, and there are no other attack method, just GCD them and found the GCD number to factor 3 n.

```
c1=187003201103675746554498235530092127249373184421011405813783589282049948274981398418974791686751237893744626370952655644721

e1=65537
n1=237957191452253868040550159459763315048788514404649567685964871677107014688170801746169235333971441406675184145169284167247
```

```
c2=279793681571708907670300690601940385261345994974568466209840542119064130244104000260536940072477735729723571065746361869873

e2=65537
n2=469140960847672389678144939972947402868380535723865027279109037949392836331979974273831965692961882995579782797324217254694



c3=240848794500152041368317447597343713506962783252273270497434347123094568088673984889157981762827696169552472765068077392494

e3=65537
n3=245430033937126927690381372230308554018353442959687171773806398980236464078074651977612115291433361050573257067882291295199


def int2text(message):
   result=""
   while message>0:
       result = chr(int(message)%int(256))+ result
       message=int(message)/int(256)
   return result

import primefac

p1=primefac.gcd(n1,n2)
q1=n1/p1
d=primefac.modinv(e1,(p1-1)*(q1-1))%((p1-1)*(q1-1))
m1=pow(c1,d,n1)
print int2text(m1)

p2=p1
q2=n2/p2
d=primefac.modinv(e2,(p2-1)*(q2-1))%((p2-1)*(q2-1))
m2=pow(c2,d,n2)
print int2text(m2)


p3=primefac.gcd(n2,n3)
q3=n3/p3
d=primefac.modinv(e3,(p3-1)*(q3-1))%((p3-1)*(q3-1))
m3=pow(c3,d,n3)
print int2text(m3)
```

400

This challenge is a white-box protocol analysis aimed to break the authentication system.

Following is the work flow of this authenticatoin system:

1. the user send a login request with username to the server
2. the server send Nonce and ChallengeCookie = Base64Encode(RandomIV | AES128-CBC(RandomIV,Nonce | U | Timestamp, KS)) back to the user
3. the user send the challenge response(R = SHA256(Nonce | P), where P is the password for authentication) to the server

   the server verify whether the password and username is right or not. if right the server will issue a ticket to user, Ticket = Base64Encode(RandomIV | AES128-CBC(RandomIV,Identity | TicketTimestamp, KS)) where Identity = JSON string: { user: U, groups: [G1, G2, ...] } where G1, G2, ... are the names of the groups that U belongs to

   the user can use the ticket to run some command, if the username in the ticket is admin, we can run the command "getflag"

to break this authentication protocol, we can send a login request with username 'AAAAAAAA' + '{"user": "admin", "groups": ["admin"]}\x00' to the server.

the server will response with Base64Encode(RandomIV | AES128-CBC(RandomIV, Nonce | 'AAAAAAAA{"user": "admin", "groups": ["admin"]}\x00 | Timestamp, KS)).

since the AES128-CBC is a block cipher with CBC mode, we can use the AES128-CBC(RandomIV, Nonce | 'AAAAAAAA) as the newIV, and the remain part will be AES128-CBC(newIV,{"user": "admin", "groups": ["admin"]}\x00 | Timestamp), which is a valid admin ticket.

then we can use the ticket to run getflag command and get the flag.

```
from pwn import *
import base64
from Crypto.Cipher import AES


io=remote("localhost",9999)

def toNullTerminatedUtf8(s):
   return unicode(s).encode("utf-8") + "\x00"
payload="\x01"+"A"*8+'{"user": "admin", "groups": ["admin"]}\x00'
io.send(payload)
data=io.recv(1000)
nounce=data[1:9]
cookie_b64=data[9:]
cookie = base64.b64decode(cookie_b64)
iv=cookie[:16]


fake_ticket=cookie[16:]
fake_ticket_b64=base64.b64encode(fake_ticket)

cmd="\x06"+fake_ticket_b64+"\x00"+"getflag\x00"
io.send(cmd)
io.interactive()
```

Reverse-Binary

100

We first find base64-encoded data from the pcap file.

```
> POST / HTTP/1.1\r\n
  Host: 192.168.107.14\r\n
  Connection: keep-alive\r\n
  Accept-Encoding: gzip, deflate\r\n
  Accept: */*\r\n
  User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0
  Content-Type: application/x-www-form-urlencoded\r\n
> Content-Length: 264\r\n
  \r\n
  [Full request URI: http://192.168.107.14/]
  [HTTP request 1/1]
  [Response in frame: 5]
  File Data: 264 bytes
v HTML Form URL Encoded: application/x-www-form-urlencoded
  v Form item: "MzU5OThmZGI3ZmUzYjc5NDBiOTM3NWE2OGE2NTRmZjk0OWM1OGRjYjliMWF1YmIwNDhkNmFhNzRkOTA1
      Key [truncated]: MzU5OThmZGI3ZmUzYjc5NDBiOTM3NWE2OGE2NTRmZjk0OWM1OGRjYjliMWF1YmIwNDhkNmFh
      Value: =
```

```
0120  3a 20 61 70 70 6c 69 63  61 74 69 6f 6e 2f 78 2d   : applic ation/x-
0130  77 77 77 2d 66 6f 72 6d  2d 75 72 6c 65 6e 63 6f   www-form -urlenco
0140  64 65 64 0d 0a 43 6f 6e  74 65 6e 74 2d 4c 65 6e   ded··Con tent-Len
0150  67 74 68 3a 20 32 36 34  0d 0a 0d 0a 4d 7a 55 35   gth: 264 ····MzU5
0160  4f 54 68 6d 5a 47 49 33  5a 6d 55 7a 59 6a 63 35   OThmZGI3 ZmUzYjc5
0170  4e 44 42 69 4f 54 4d 33  4e 57 45 32 4f 47 45 32   NDBiOTM3 NWE2OGE2
0180  4e 54 52 6d 5a 6a 6b 30  4f 57 4d 31 4f 47 52 6a   NTRmZjk0 OWM1OGRj
0190  59 6a 6c 69 4d 57 46 6c  59 6d 49 77 4e 44 68 6b   YjliMWFl YmIwNDhk
01a0  4e 6d 46 68 4e 7a 52 6b  4f 54 41 31 59 6a 64 69   NmFhNzRk OTA1Yjdi
01b0  4d 47 4d 32 5a 54 41 30  59 6a 51 77 4e 47 56 69   MGM2ZTA0 YjQwNGVi
01c0  4e 6a 45 78 4d 6a 6c 6d  4f 54 4a 68 5a 44 6b 78   NjExMjlm OTJhZDkx
01d0  4d 6a 63 77 4d 7a 67 31  4d 44 49 77 4d 54 55 34   MjcwMzg1 MDIwMTU4
01e0  4d 6d 4e 6c 4d 7a 6c 6c  4e 7a 64 69 5a 6d 55 33   MmNlMzll NzdiZmU3
01f0  4d 7a 6c 6d 5a 57 4d 31  4d 6a 67 33 4e 44 46 69   MzlmZWM1 Mjg3NDFi
0200  4d 6a 41 79 5a 6a 67 35  4d 6a 4e 68 4f 57 59 34   MjAyZjg5 MjNhOWY4
0210  5a 44 59 7a 4d 44 4d 32  4d 54 64 6b 4f 47 55 32   ZDYzMDM2 MTdkOGU2
0220  5a 54 4d 31 59 54 42 6b  4e 6a 51 30 4d 54 45 31   ZTM1YTBk NjQ0MTE1
0230  5a 54 49 7a 4f 44 55 79  4d 6d 4d 32 5a 44 42 6a   ZTIzODUy MmM2ZDBj
0240  59 57 4e 6b 4d 57 46 6d  5a 47 46 6c 4d 6a 4d 77   YWNkMWFm ZGFlMjMw
0250  4e 54 41 30 4e 54 4a 6a  4f 54 6b 34 5a 54 4d 35   NTA0NTJj OTk4ZTM5
```

Then, reverse the pyinstaller binary and modify the script to solve the challenge.

```
import struct, os, time, threading, urllib, requests, ctypes, base64
from Cryptodome.Cipher import AES, ARC4
from Cryptodome.Hash import SHA

infile = 'flag'
encfile = 'orig.CRYPTED'
keyfile = 'keyfile'
sz = 1024
bs = 16

def decrypt_request():
   pcap_req = "35998fdb7fe3b7940b9375a68a654ff949c58dcb9b1aebb048d6aa74d905b7b0c6e04b404eb61129f92ad912703850201582ce39e77bfe7
   _hash_chksum = pcap_req[:40]
   _hash_content = pcap_req[40:]
   dec = ARC4.new(_hash_chksum.decode('hex'))
   return dec.decrypt(_hash_content.decode('hex'))
   # 'id=d1&key=2f87011fadc6c2f7376117867621b606&iv=95bc0ed56ab0e730b64cce91c9fe9390'

def generate_keyfile():
   # n = hex(ord(id) + bs)
   n = hex(ord('d1'.decode('hex')) + 16)
   iv = "95bc0ed56ab0e730b64cce91c9fe9390".decode('hex')
   key = "2f87011fadc6c2f7376117867621b606".decode('hex')
```

```
   key = ''.join((chr(ord(x) ^ int(n, 16)) for x in key))
   iv = ''.join((chr(ord(y) ^ int(n, 16)) for y in iv))
   keyfile = open("keyfile", "wb")
   keyfile.write(key + iv)
   keyfile.close()
   print(n, iv, key)
   return True

def decrypt():
   global keyfile
   key = ''
   iv = ''
   if not os.path.exists(encfile):
       exit(0)
   while True:
       time.sleep(10)
       if os.path.exists(keyfile):
           keyin = open(keyfile, 'rb')
           key = keyin.read(bs)
           iv = keyin.read(bs)
           if len(key) != 0 and len(iv) != 0:
               aes = AES.new(key, AES.MODE_CBC, iv)
               fin = open(encfile, 'r')
               fsz = struct.unpack('<H', fin.read(struct.calcsize('<H')))[0]
               fout = open(infile, 'w')
               fin.seek(2, 0)
               while True:
                   data = fin.read(sz)
                   n = len(data)
                   if n == 0:
                       break
                   decrypted = aes.decrypt(data)
                   n = len(decrypted)
                   if fsz > n:
                       fout.write(decrypted)
                   else:
                       fout.write(decrypted[:fsz])
                   fsz -= n

               fin.close()
               os.remove(encfile)
               break

print(decrypt_request())
generate_keyfile()
decrypt()

# ----Trend Microt CTF 2018. Flag for this challenge is: TMCTF{MJB1200}
```

200

A 32-bit shellcode injector using IAT hook. `DragQueryFileW` in `notepad.exe` is hooked. The shellcode is written into `.text` section of `shell32.dll`.

```
from ida_bytes import get_bytes, patch_bytes
buf = bytearray(get_bytes(0x4031A0, 376))
for i in xrange(len(buf)):
   buf[i] ^= [0xDE, 0xAD, 0xF0, 0x0D][i % 4]
patch_bytes(0x4031A0, str(buf))
```

Run 32-bit notepad.exe and drop a file named "zdi_ftw", the rot13-enctypted flag is shown.

```
TMCTF{want_sum_iat_hooking}
```

300

The PE file is packed but it's easy to unpack. It detects debugger by `IsDebuggerPresent`, and Virtual Machine by checking the presense of specific .sys file. Checks whether the hour field of current time is 5.

```
TMCTF{F14g1s::____1G}
```

400

A river crossing puzzle with a servant(7), a dog(4), a father(6), a mother(5), two sons(2,3) and two daughters(0,1).

```
sidev = 1
def side():
   global sidev
   t = sidev
   sidev = 0 if sidev else 1
   return t

def a(x, y = None):
   if(y == None):
       return "\xD0" + chr(side()) + chr(1 << x)
   else:
       return "\xD1" + chr(side()) + chr(1 << x) + chr(1 << y)

s = a(7, 4) + a(7)
s += a(7, 0) + a(7, 4)
s += a(5, 1) + a(5)
s += a(6, 5) + a(6)
s += a(7, 4) + a(5)
s += a(6, 5) + a(6)
s += a(6, 2) + a(7, 4)
s += a(7, 3) + a(7)
s += a(7, 4)
print(s.encode("hex").lower())
```

Reverse-Other

100

Run the binary directly, it ouput "flag is here", but run the binary under debugger, it output "nice try, punk".

so it seems like there are anti-dbg techniques deployed on this binary.

No worry, there are various ways to patch the binary and bypass the anti-debugging. once we find the function that output "flag is here", we found the flag. since the flag is reside in that function.

200

Simply unpack upx, some weird strings appear in main function.
I noticed this program will self-open and read all data in memory by using API monitor.
After some reversing , I found that it will compare the section name in memory.
Simply modify the code , let program compare the encrypted flag string with section name, you got flag in debugger instantly.

400

This Challenge is a python bytecode reversing challenge.

```
import sys


def verify_flag(flag): pass
verify_flag.__code__ = verify_flag.__code__.__class__( 1 , 20 , 9 , 67 , 'y\x0c\x00|\x00\x00d\x01\x00\x17\x01Wn%\x00\x01\x01\x

if __name__ == "__main__":
   if len(sys.argv) != 2:
       print "Usage:"
       print "    %s flag" % sys.argv[0]
   else:
       if verify_flag(sys.argv[1]):
           print "%s is the correct flag!" % sys.argv[1]
       else:
           print "Better luck next time"
```

Seems like we have to reverse the python bytecode verify_flag.code.

We try to decompile the bytecode by crafting a pyc file and use uncompyle6 to decompile, but it doesn't work since the code contains non-ascii characters.

Finally, We sucessfully decompiled the bytecode by craft a Code2 object and call uncompyle6.main.decompile() directly

```
import xdis
import sys
from xdis.code import Code2
from xdis.bytecode import get_instructions_bytes
import uncompyle6


argcount = 1
nlocals = 20
stacksize = 9
flags = 67
code = b'y\x0c\x00|\x00\x00d\x01\x00\x17\x01Wn%\x00\x01\x01\x01x9\x00|\x00\x00D]\x10\x00}\x01\x00|\x01\x00|\x01\x007}\x01\x00q
consts = (None, 0, 'TMCTF{', '}', 1, -1, 7, 5, 'ReadEaring', 'adEa', 'dHer', 24, 9, 'h', 255, 8, 32, '', 'R) +6', 'l1:C(', ' R
names = ('True', '\xe0\xa1\xb5\xe0\xa1\xb5HA', 'len', 'False', 'startswith', 'endswith', 'replace', 'split', 'rsplit', 'Assert
varnames = ('inval', 'c', 'l', 's', 'sdl', 'x', 'ROFL', 'KYRYK', 'QQRTQ', 'KYRYJ', 'QQRTW', 'KYRYH', 'QQRTE', 'KYRYG', 'QQRTR'
filename = 'flag.py'
name = 'verify_flag'
firstlineno = 1337
lnotab = '\x00\x01\x03\x01\x0c\x01\x03\x01\r\x01\x0e\x02\x07\x02\t\x01\x0e\x02\x03\x02\x03\x01\x08\x01\x03\x01\x04\x02\x12\x01

co = Code2(argcount, 0, nlocals, stacksize, flags, code, consts, names, varnames, filename, name, firstlineno, lnotab, (), ())
version = 2.7
timestamp = 1536287532
code_objects = {co: co}
source_size = None
is_pypy = False
magic_int = 62211
uncompyle6.main.decompile(version, co, sys.stdout, None, False, timestamp, False, code_objects=code_objects, source_size=sourc
```

following is the decompiled code

```
# uncompyle6 version 3.2.3
# Python bytecode 2.7 (62211)
# Decompiled from: Python 3.7.0 (default, Jul 15 2018, 10:44:58)
# [GCC 8.1.1 20180531]
# Embedded file name: flag.py
# Compiled at: 2018-09-07 10:32:12
try:
   inval + 0
except:
   for c in inval:
       c += c
   else:
       del c

else:
   while 1:
       if True:
           inval += inval
   else:
       del inval

try:
   à¡µà¡µHA
except:
   pass

if len(inval) == 0 or False:
   return False
if not inval.startswith('TMCTF{'):
   return False
if not inval.endswith('}'):
   return False
   inval = inval.replace('TMCTF{')
else:
```

```
    l = len(inval)
    inval = inval.split('TMCTF{', 1)[-1].rsplit('}', 1)[0]
    try:
        assert len(inval) + 7 == l
    except:
        return False

    10
if inval == ('ReadEaring').replace('adEa', 'dHer'):
    return False
inval = map(ord, inval)
l = len(inval)
if l != 24:
    return False
s = sum(inval)
if s % l != 9:
    return False
sdl = s / l
if chr(sdl) != 'h':
    return False
inval = [ x ^ sdl for x in inval ]
ROFL = list(reversed(inval))
KYRYK = [0] * 5
QQRTQ = [0] * 5
KYRYJ = [0] * 5
QQRTW = [0] * 5
KYRYH = [0] * 5
QQRTE = [0] * 5
KYRYG = [0] * 5
QQRTR = [0] * 5
KYRYF = [0] * 5
QQRTY = [0] * 5
for i in xrange(len(KYRYK)):
    for j in xrange(len(QQRTQ) - 1):
        KYRYK[i] ^= inval[i + j]
        if QQRTQ[i] + inval[i + j] > 255:
            return False
        QQRTQ[i] += inval[i + j]
        KYRYJ[i] ^= inval[i * j]
        if QQRTW[i] + inval[i * j] > 255:
            return False
        QQRTW[i] += inval[i * j]
        KYRYH[i] ^= inval[8 + i * j]
        if QQRTE[i] + inval[8 + i * j] > 255:
            return False
        QQRTE[i] += inval[8 + i * j]
        KYRYG[i] ^= ROFL[8 + i * j]
        if QQRTR[i] + ROFL[8 + i * j] > 255:
            return False
        QQRTR[i] += ROFL[8 + i * j]
        KYRYF[i] ^= ROFL[i + j]
        if QQRTY[i] + ROFL[i + j] > 255:
            return False
        QQRTY[i] += ROFL[i + j]

    KYRYK[i] += 32
    KYRYJ[i] += 32
    KYRYH[i] += 32
    KYRYG[i] += 32
    KYRYF[i] += 32
    QQRTE[i] += 8
    QQRTY[i] += 1

for ary in [KYRYK, KYRYJ, KYRYH, KYRYG, KYRYF, QQRTW, QQRTE, QQRTR, QQRTY]:
    for x in ary:
        if x > 255:
            return False

if ('').join(map(chr, KYRYK)) != 'R) +6':
```

```
   return False
try:
   if ('').join(map(chr, QQRTQ)) != 'l1:C(':
       return False
except ValueError:
   return False

if ('').join(map(chr, KYRYJ)) != ' RP%A':
   return False
if tuple(QQRTW) != (236, 108, 102, 169, 93):
   return False
if ('').join(map(chr, KYRYH)) != ' L30Z':
   print 'X2'
   return False
if ('').join(map(chr, QQRTE)) != ' j36~':
   print 's2'
   return False
if ('').join(map(chr, KYRYG)) != ' M2S+':
   print 'X3'
   return False
if ('').join(map(chr, QQRTR)) != '4e\x9c{E':
   print 'S3'
   return False
if ('').join(map(chr, KYRYF)) != '6!2$D':
   print 'X4'
   return False
if ('').join(map(chr, QQRTY)) != ']PaSs':
   print 'S4'
   return False
return True
```

By reversing the decompiled code, we realized that it is very easy to get the flag by z3 solver.

```
from z3 import *

flag = []
constraints = []
sum_flag=2505
for i in range(24):
  flag.append(BitVec('x%d' % i, 16))
  constraints.append(flag[i]<0x7f)
  constraints.append(flag[i]>0x20)

  sum_flag-=flag[i]
constraints.append(sum_flag==0)

flag_enc = [x ^ 104 for x in flag]
flag_enc_rev = list(reversed(flag_enc))

aa = [0] * 5
bb = [0] * 5
cc = [0] * 5
dd = [0] * 5
ee = [0] * 5
ff = [0] * 5
gg = [0] * 5
hh = [0] * 5
ii = [0] * 5
jj = [0] * 5
for i in range(len(aa)):
  for j in range(len(bb) - 1):
      aa[i] ^= flag_enc[i + j]
      #if bb[i] + flag_enc[i + j] > 255:
      #    return False
      bb[i] += flag_enc[i + j]
      cc[i] ^= flag_enc[i * j]
      #if dd[i] + flag_enc[i * j] > 255:
      #    return False
      dd[i] += flag_enc[i * j]
```

```
      ee[i] ^= flag_enc[8 + i * j]
      #if ff[i] + flag_enc[8 + i * j] > 255:
      #    return False
      ff[i] += flag_enc[8 + i * j]
      gg[i] ^= flag_enc_rev[8 + i * j]
      #if hh[i] + flag_enc_rev[8 + i * j] > 255:
      #    return False
      hh[i] += flag_enc_rev[8 + i * j]
      ii[i] ^= flag_enc_rev[i + j]
      #if jj[i] + flag_enc_rev[i + j] > 255:
      #    return False
      jj[i] += flag_enc_rev[i + j]

  aa[i] += 32
  cc[i] += 32
  ee[i] += 32
  gg[i] += 32
  ii[i] += 32
  ff[i] += 8
  jj[i] += 1

#for ary in [aa, cc, ee, gg, ii, dd, ff, hh, jj]:
#    for x in ary:
#        if x > 255:
#            return False

compare = list(map(ord, 'R) +6'))
for i in range(5):
  constraints.append(aa[i] == compare[i])

compare = list(map(ord, 'l1:C('))
for i in range(5):
  constraints.append(bb[i] == compare[i])

compare = list(map(ord, ' RP%A'))
for i in range(5):
  constraints.append(cc[i] == compare[i])

compare = (236, 108, 102, 169, 93)
for i in range(5):
  constraints.append(dd[i] == compare[i])

compare = list(map(ord, ' L30Z'))
for i in range(5):
  constraints.append(ee[i] == compare[i])

compare = list(map(ord, ' j36~'))
for i in range(5):
  constraints.append(ff[i] == compare[i])

compare = list(map(ord, ' M2S+'))
for i in range(5):
  constraints.append(gg[i] == compare[i])

compare = list(map(ord, '4e\x9c{E'))
for i in range(5):
  constraints.append(hh[i] == compare[i])

compare = list(map(ord, '6!2$D'))
for i in range(5):
  constraints.append(ii[i] == compare[i])

compare = list(map(ord, ']PaSs'))
for i in range(5):
  constraints.append(jj[i] == compare[i])

#print(constraints)
print(solve(constraints))
```

Forensics-crypto1

100

25 x 25 qr-code version 2

type infomation bits: 010101111x101101, ECC level: Q, mask: 7

(((row + column) mod 2) + ((row * column) mod 3)) mod 2 == 0

Error corrections available but data is removed (right side)

This is readed as follows
1001110110001001110110001001010001101110010001101110010001101011110101000010111101011000100111010011001100101101010011100011000101

QR CODE has many modes, we try to use byte mode to deocode it.(4 bit mode, 8 bit length, the size of per data block is 8)

By trying to decode the QR CODE manually, we can find that the flag is ended with N1nj4}. So the length of this string is 0b00010100.

The Data and ECC block can be read under the rule as follows. But we need to read it from offset 4 + 8.

Due to the fact that the ecc level of this QR CODE is %25, we just need to patch some known letters as “TWCTF”,etc.

But we need to know how to patch it.
XORed

we need to patch the header of the QR CODE, mode and lemgth and then we need to XOR them with mask.
After patching known bytes, we can scan the QR CODE.

flag is here
TMCTF{QRc0d3-N1nj4}

200

Decompiling pyinstaller shows the sourcecode.

```
$ cat OceanOfSockets.py

...
def request():
   try:
       connection = httplib.HTTPConnection(sys.argv[1], sys.argv[2])
       connection.request('GET', '/tmctf.html')
       resTMCF = connection.getresponse()
       readData = resTMCF.read()
       if 'OceanOfSockets' in readData:
           headers = {'User-Agent': 'Mozilla Firefox, Edge/12',
            'Content-type': 'text/html',
            'Cookie': '%|r%uL5bbA0F?5bC0E9b0_4b2?N'}
           connection.request('GET', '/index.html', '', headers)
       else:
           sys.exit(0)
   except:
       pass
...
```

There doesn't seem to be much information except the suspicious cookie.

While thinking about the flag format (which is `TMCTF{}`), I realized it should be a simple addition algorithm used on Cookie.

```
>>> [chr((ord(i) + 47)) for i in '%|r%uL5bbA0F?5bC0E9b0_4b2?N']
['T', '\xab', '\xa1', 'T', '\xa4', '{', 'd', '\x91', '\x91', 'p', '_', 'u', 'n', 'd', '\x91', 'r', '_', 't', 'h', '\x91', '_',
```

Now it sounds like some of characters are not displayed properly. I decided to mod a byte to leak remaining ambiguous bytes.

```
>>> [chr((ord(i) + 47) % 0x5e) for i in '%|r%uL5bbA0F?5bC0E9b0_4b2?N']
['T', 'M', 'C', 'T', 'F', '\x1d', '\x06', '3', '3', '\x12', '\x01', '\x17', '\x10', '\x06', '3', '\x14', '\x01', '\x16', '\n',
```

Merging above results will print the flag

flag: `TMCTF{d33p_und3r_th3_0c3an}`

400

We got these informations from challenge:

```
n = 144 and l = 288
fi (x) = x XOR ki
unknown h
```

We known a couple of plaintext/cipher, so it's known plaintext attack.

Here the F function of feistel is XOR, all differences are transmitted by probability 1. So we can decrypt all cipher over these conditions.

Fistly, we do some pre-works:

L	R
R	R^L^K1
R^L^K1	L^K1^K2
L^K1^K2	R^K2^K3
R^K2^K3	R^L^K1^K3^K4
R^L^K1^K3^K4	L^K1^K2^K4^K5
...	...

we don't know h, so we need to try alot of time, and finally I found:

```
true_l=xor(lm,xor(rc,r))
```

We guess the h=5 or like 5 round's result, so we can solve it:

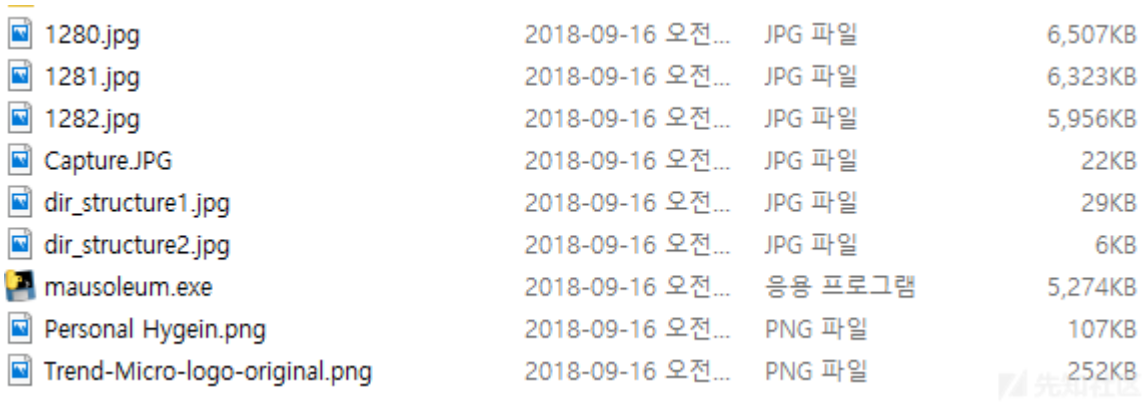
```
m1="0100000101101110001000000110000101110000011100000110110001100101001000000110000101101110011001000010000001100001011011100

c1="0001001000110001011101010011011001100011001100010011101000111101011000000011110010010111000110011001110000000011010010010101

c="0000001100001110010111000010000000011001001011000001001001111110000010010000011000000011000001001000100100010011101001010011
```

```
def xor(bin1,bin2):
   assert len(bin1)==len(bin2)
   s=""
   for i in range(len(bin1)):
       s+=str(int(bin1[i])^int(bin2[i]))
   return s
from Crypto.Util.number import long_to_bytes
def show(x):
   print long_to_bytes(int(x,2))

lm=m1[0:144]
rm=m1[144:]
lc=c1[0:144]
rc=c1[144:]

l=c[0:144]
r=c[144:]



true_l=xor(lm,xor(rc,r))
true_r=xor(xor(xor(xor(lc,l),true_l),lm),rm)

show(true_l+true_r)
```

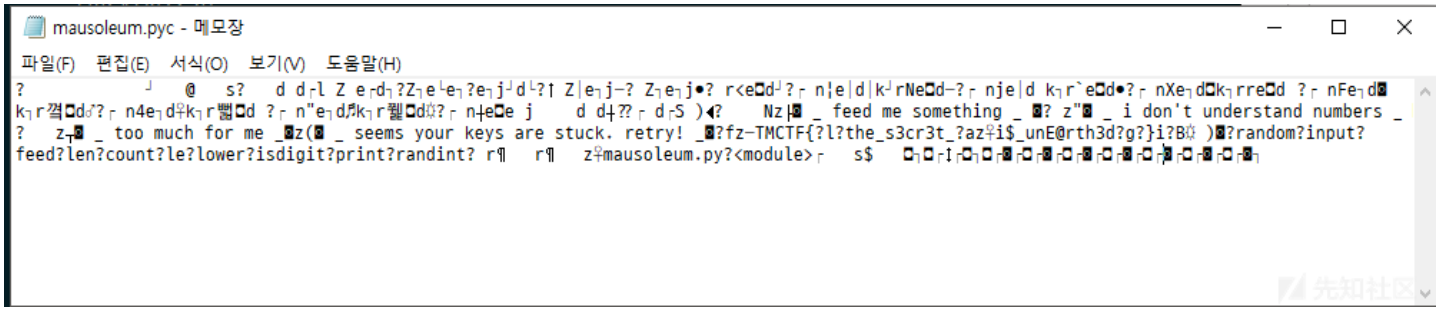
Forensics-crypto2

200

Dump HTTP upload requests and you will get mausoleum.exe.



From there, decompile the exe file (omg so many pyinstaller binary)



I was somehow unable to decrypt the python file (I changed some of bits in headers, still it didn't work)

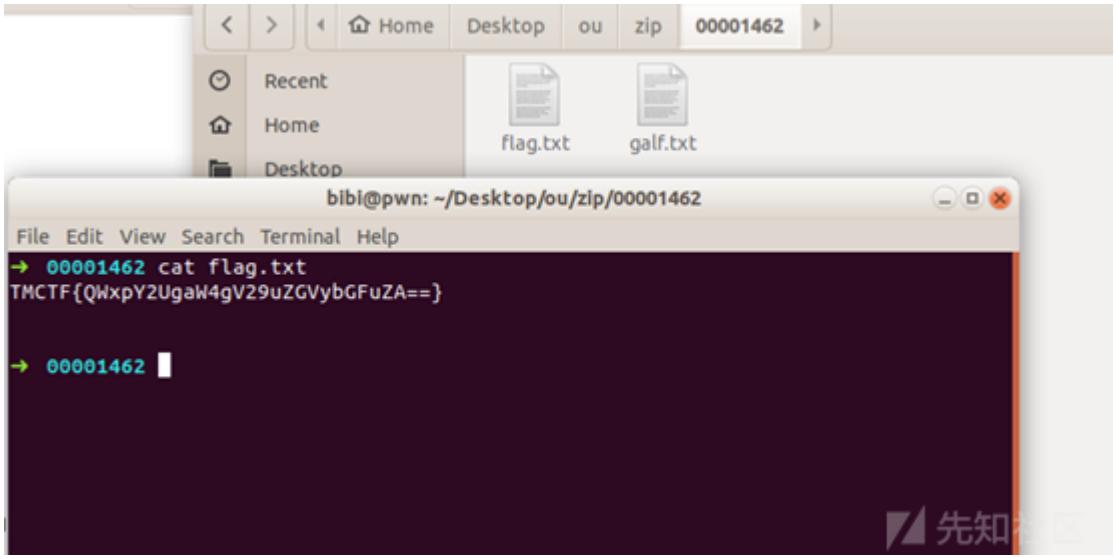
so I decided to remove useless letters from the notepad and get the flag. Guess what? I successfully submitted it on the first guess.

`TMCTF{the_s3cr3t_i$_unE@rth3d}`

Misc

100

Foremost extract a zip file and unzip it to get flag:

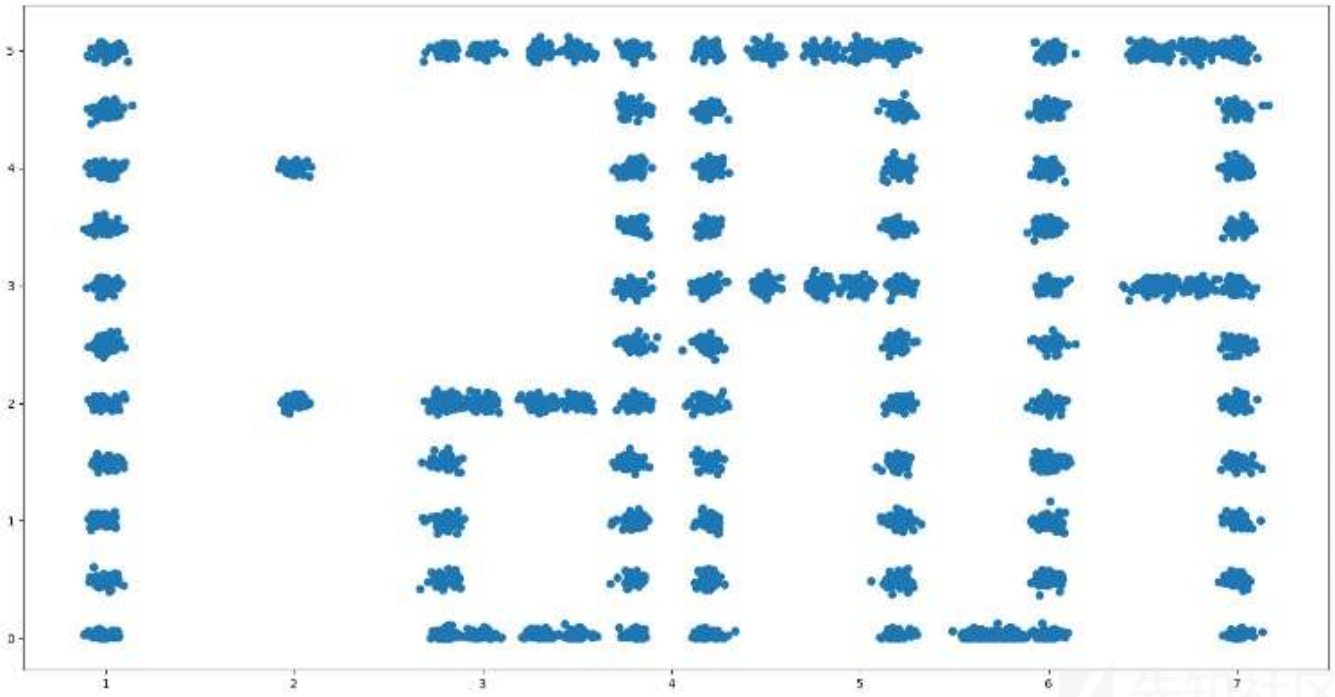


200

The challenges provides a pcap file and a python script. The python script reads a txt file into an array and uses it as training data for DBSCAN. So I guess the purpose of this challenge is to extract the data from the pcap.

I use `strings` to observe that the data of icmp packet looks like the data we want ,so I extract them with the following command
`tshark -r ./traffic.pcap -Y "icmp and ip.src_host==192.168.0.17" -T fields -e data`

Decode them and apply them to the python script. But the model outputs nothing. So I decide to plot it directly.



And the flag is `FLAG:1`

上一篇：从一道ctf题目学到的绕过长度执行... 下一篇：Android OS 中通过 Wi...

1. 0 条回复
   - 动动手指，沙发就是你的了！