1337pwnie Add original challenge — Latest commit 5b289cf 7 days ago

..

README.md	Style fixes	7 days ago
backtrack.py	Add Python scripts	7 days ago
decrypt.py	Add Python scripts	7 days ago
xref.zip	Add original challenge	7 days ago

README.md

xref - 300 points

- Cryptography

Problem

Eric was working on a reversing challenge last night when his computer was struck by ransomware! Can you decrypt enough of the files to recover the print_flag binary?

Solution

Reconnaissance

We are given a zip archive containing a directory with some files:

```
Archive:  xref.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2017-04-18 16:01   final/
    15948  2017-04-18 16:01   final/Archive.zip.xor
   561782  2017-04-18 16:01   final/bae.gif.xor
   719159  2017-04-18 16:01   final/beemovie.whitespace.xor
      173  2017-04-18 16:01   final/decrypt_diary.py.xor
    17312  2017-04-18 16:01   final/diary.ecb.xor
    18015  2017-04-18 16:01   final/fire_meme.jpg.xor
    71549  2017-04-18 16:01   final/lossless_meme.png.xor
      191  2017-04-18 16:01   final/motivational.txt.xor
     8776  2017-04-18 16:01   final/print_flag.xor
      170  2017-04-18 16:01   final/protect_diary.py.xor
      993  2017-04-18 16:01   final/ransomware.py
      595  2017-04-18 16:01   final/README.txt
    19037  2017-04-18 16:01   final/special_meme.jpg.b64.xor
   312855  2017-04-18 16:01   final/wikipedia_fair_use_sample.ogg.xor
---------                     -------
  1746555                     15 files
```

The README file states that all of the files on the hard drive are encrypted using a "perfectly secure one-time pad", and the accompanying ransomware code is also supplied. Judging from the filename, the print_flag file should print the flag.

The ransomware encrypts all of the files using the same 'one time pad'. Reusing a one time pad utterly destroys most of the secrecy that the pad provides, and we can mount a many-time-pad attack against it.

To do this, we make use of the fact that the same (unknown) pad is xorred with all of the files. If we know (or guess) even one byte of one file in the directory, we can recover that byte (that is, the byte with the same offset) of *any* of the other files.

Because any byte has been xorred with the key, we xor the (known or guessed) byte with the result to recover part of the key. If we xor this part of the key with any of the other files, we recover the plaintext.

Ideally, we would be able to find the plain version of one of the larger files, which would immediately enable us to decrypt the print_flag file. The file names seemed to suggest it should be possible to find a copy of one of them somewhere out on the Internet. However, searching around did not yield any results. Next up: try to recover the key the hard way.

First part: manual recovery

With a quick custom script we can partly automate the process, providing operations for decrypting a file using the (partly) known key, recovering part of the key using a xorred file and its accompanying plaintext, and adding a few bytes to the key by providing a xorred file and the next few plaintext bytes:

```
#!/usr/bin/python2
import sys


def main():
    otp = open('otp', 'rb').read()
    if sys.argv[1] == 'add':
        otpw = open('otp', 'ab')
        ciphertext = open(sys.argv[2], 'rb').read()
        add_plain = sys.argv[3]
        for l, r in zip(ciphertext[len(otp):], add_plain):
            otpw.write(chr(ord(l) ^ ord(r)))
    elif sys.argv[1] == 'set':
        otpw = open('otp', 'wb')
        ciphertext = open(sys.argv[2], 'rb').read()
        plaintext = open(sys.argv[3], 'rb').read()
        sys.stderr.write('{} {}\n'.format(len(ciphertext), len(plaintext)))
        for l, r in zip(ciphertext, plaintext):
            otpw.write(chr(ord(l) ^ ord(r)))
    else:
        ciphertext = open(sys.argv[1], 'rb').read()
        sys.stderr.write('{} {}\n'.format(len(ciphertext), len(otp)))
        plaintext = [
            chr(ord(l) ^ ord(r))
            for l, r in zip(ciphertext, otp)]
        sys.stdout.write(''.join(plaintext))

if __name__ == '__main__':
    main()
```

The first few bytes could be recovered fairly easily since the jpg and gif files have a static header. After that, some of the bytes could be manually recoveredby going back and forth between the .txt.xor and py.xor files, each time guessing some of the following bytes by looking at the text and code structure. After recovering a few words of the motivational.txt file, a Google search revealed it to contain the text of a Twitter post from the challenge author. Well, almost. The text had been altered slightly, but having a better idea of what it should look like did speed up the recovery process and easily produced the content of the python files, which could encrypt and decrypt the diary.

Second part: automating the process

The python files show that the diary has been encrypted using AES in ECB mode:

```
from Crypto.Cipher import AES
c=AES.new('theymustnevrknow',AES.MODE_ECB)
d=open('diary.txt','r').read()
d=d+"i"*(16-len(d)%16)
open('diary.ecb','wb').write(c.encrypt(d))
```

This can be used to automate the decryption process, combined with the following file properties:

- `beemovie.whitespace.xor` only contains whitespace characters: `0x20` , `0x09` , and `0x0a0d`
- `special_meme.jpg.b64.xor` only contains base64 characters: `[a-zA-Z0-9+/]`
- `diary.ecb.xor` contains plaintext ascii characters encrypted in 16-byte blocks So the key could be recovered using the following code:

```
#!/usr/bin/python3
import string

from Crypto.Cipher import AES

DICT = [[0x20], [0x09], [0x0a, 0x0d]]
BEE = open('beemovie.whitespace.xor', 'rb').read()
B64 = open('special_meme.jpg.b64.xor', 'rb').read()
DIARY = open('diary.ecb.xor', 'rb').read()
B64_DICT = set(str.encode(string.ascii_letters + string.digits + '+/'))
DIARY_DICT = set(str.encode(string.printable))
DIARY_DICT_STRICT = set(open('diary_d.txt', 'rb').read())
CIPHER = AES.new('theymustnevrknow', AES.MODE_ECB)
OTP = open('otp', 'rb').read()


def valid_diary(otp_bytes, base_pos):
    remainder = len(OTP) % 16
    otp = OTP[:len(OTP)-remainder] + otp_bytes[:16]
    result = bytes([
        c1 ^ c2
        for c1, c2 in zip(DIARY, otp)])
    data = CIPHER.decrypt(result)
    #if all([char in DIARY_DICT for char in data[-16:]]):
    if all([char in DIARY_DICT_STRICT for char in data[-16:]]):
        print(data)
        return True
    return False


def valid_b64_byte(otp_byte, pos):
    return otp_byte ^ B64[pos] in B64_DICT


def backtrack(cur_bytes, base_pos):
    if len(cur_bytes) >= 16:
        if valid_diary(cur_bytes, base_pos):
            yield cur_bytes
            return
        else:
            return

    for chars in DICT:
        valid = True
        new_bytes = cur_bytes
        for char in chars:
            pos = base_pos+len(new_bytes)
            new_bytes += (BEE[pos] ^ char).to_bytes(1, byteorder='big')
            if not valid_b64_byte(new_bytes[-1], pos):
                valid = False
        if valid:
            for result in backtrack(new_bytes, base_pos):
                yield result


def main():
    global OTP
    pos = 0
    while pos < min([len(BEE), len(B64), len(DIARY)]):
        remainder = len(OTP) % 16
        results = [
            next(backtrack(OTP[len(OTP)-remainder:], len(OTP)-remainder))]
        OTP += results[0][remainder:]
        print('len otp ', len(OTP))
        open('otp_calculated', 'wb').write(OTP)


if __name__ == '__main__':
    main()
```

When some more bits of the key were decrypted this way, it was possible to extract low-res versions of some of the images. After repairing those using Imagemagick convert, and throwing them at Google image search, the original ones were revealed - all from the Twitter timeline of the challenge author. However, the images seemed to be re-encoded and thus useless for speeding up key recovery. After running the code for an hour or so, enough bytes of the key were recovered to be able to decrypt the print_flag file.

Finally, success and some disappointment

I only had a 32-bit virtual machine at the time, and the binary was 64 bits. After downloading and installing a 64-bit VM and running the binary, the key was recovered:

```
This crib drag is treacherous.
flag{n0th1_saf3_i5_w0rth_th3_dr1ve}
```

Unfortunately, I did not manage to submit the flag since the CTF time limit was reached just when I had managed to install the new VM :-/