Delusions of Grandeur (NetSec) Blog

Tuesday, September 24, 2013

CSAW CTF Quals: Cryptography 300 Onlythisprogram

For this challenge, we are given the onlythisprogram.tar.gz file. We extracted the files and in that file was the onlythisprogram.py script and an output file with 9 fileX.enc files. Looking through the python code, this line is of interest:

```
args.outfile.write(chr(ord(keydata[counter % len(keydata)]) ^ ord(byte)))
```

This line shows that the operation of interest is an XOR operation. Here is a reading on that vulnerability:

The first thing that I thought might have been it was crib dragging, exactly like it showed in the reading. Still brainstorming, however, I guessed that it probably wasn't just text files, but the ENC files were the encrypted files.

Okay, still looking at the same line we have above, the modulus is happening with the length of the keydata. Further above, you see this:

```
while (args.secretkey.tell() < blocksize):
```

Blocksize is set to 256 at the top so you know that the secretkey file is a 256 byte key that is used over and over again until the files are done being encrypted.

Great! Now it's time to build the decryption script.

I want the script to take the bytes from two files and then XOR them together and output that to an outfile. This is pretty much a straight Ctrl+C, Ctrl+V from the CSAW provided python file! #EPIC!

So, if each fileX.enc represented a file, then they would probably have headers that we know are at the beginning. This is the library that I used for the file headers.

So now you need to understand what cribbing does. Here's an example. If you have encrypted byte AE and encrypted byte BE, both of which use the same key (K), you can XOR AE and BE together. What that does is it gets rid of the key and gives you A ^ B, where A and B are the original bytes. At that point, if you XOR that with the original byte A, you will get the original byte of B.

So for this particular challenge, if we suspect that the file1 is supposed to be a DOC file, for example, we can assume the original byte in that position should be the header byte of a DOC file. If the bytes returned is not gibberish, then we know that one of file1 or whatever file we XOR'd it against to be a DOC file. Here's the code for this function:

```
header=open('header','w+')
header.write('\xFF\xD8\xFF\xE0\x00\x10\x4A\x46\x49\x46\x00')
header.seek(0)

...

while 1:
    byte1, byte2 = args.infile1.read(1), args.infile2.read(1)
    if not byte1 or not byte2:
        break
    temp.write(chr(ord(byte1) ^ ord(byte2)))

temp.flush()
temp.seek(0)

while 1:
    byte1, byte2 = args.infile1.read(1), temp.read(1)
    if not byte1 or not byte2:
        break
    args.outfile.write(chr(ord(byte1) ^ ord(byte2)))
```

I did this with file0 and ran it against the other files until something gave me another header set that was in the website that I showed you. File0 was not a doc file, as most other files gave me gibberish in return, but one of the files was. I then took that file and ran the same operation against all the other files and was able to find all the file types.

file0 - MID, MIDI
file1, 3 - JFIF, JPE, JPEG, JPG
file2 - PNG
file4 - GZ, TGZ
file5 - BMP, DIB
file6 - GIF
file7 - DOC, DOT, PPS, PPT, XLA, XLS, WIZ
file8 - PDF, FDF

The longest header available to us is the JPG file type. After that, I was able to use trailer bytes to find out with certainty 4 more bytes. At this point, I was a little stumped so I downloaded a lot of files matching the file types I needed. Eventually, I realized that BMP files (at least in the ones I downloaded) had several large blocks of FF bytes. (Read using Hex Editor - I used HxD)

So using a byte of FF for the original message, I ran that against file5. Hopefully, I would get a large chunk of the secretkey.dat file. Maybe even the whole thing! Here's the code to do that:

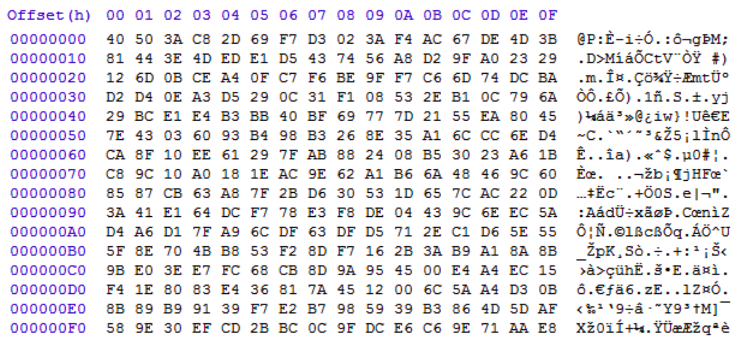
```
while 1:
    byte1, byte2 = args.infile1.read(1), '\xff'
    if not byte1 or not byte2:
        break
    args.outfile.write(chr(ord(byte1) ^ ord(byte2)))
```

I named this file buzzbmp and went through every 256 byte block until I found one that matched up with my first 11 byte block. I copy and pasted that using HxD to a new secret.dat file and ran the original onlythisprogram.py script with the following code commented out:

```
#args.secretkey.truncate()
```

```
#while (args.secretkey.tell() < blocksize):
# maybe remove the next line for release since it makes it more obvious the key only generates
#       sys.stdout.write('.')
#       args.secretkey.write(os.urandom(1))
```

The file to decrypt I used was the PDF file, file8. After several iterations of error checking for correct and incorrect bytes, a decrypted PDF file that had a few errors. However, I had enough correct bytes in that secret.dat test file that a lot of strings were intact. I had about two bytes that were off. Here is that secret.dat file:

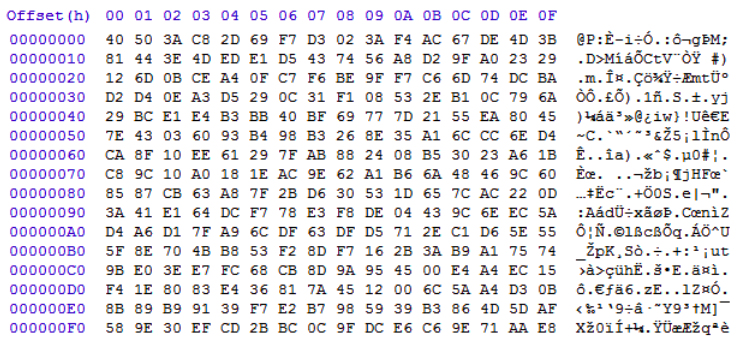


```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  40 50 3A C8 2D 69 F7 D3 02 3A F4 AC 67 DE 4D 3B  @P:È-i÷Ó.:ô¬gÞM;
00000010  81 44 3E 4D ED E1 D5 43 74 56 A8 D2 9F A0 23 29  .D>MíáÕCtV¨ÒŸ #)
00000020  12 6D 0B CE A4 0F C7 F6 BE 9F F7 C6 6D 74 DC BA  .m.Î¤.Çö¾Ÿ÷ÆmtÜº
00000030  D2 D4 0E A3 D5 29 0C 31 F1 08 53 2E B1 0C 79 6A  ÒÔ.£Õ).1ñ.S.±.yj
00000040  29 BC E1 E4 B3 BB 40 BF 69 77 7D 21 55 EA 80 45  )¼áä³»@¿iw}!Uê€E
00000050  7E 43 03 60 93 B4 98 B3 26 8E 35 A1 6C CC 6E D4  ~C.`"´˜³&Ž5¡lÌnÔ
00000060  CA 8F 10 EE 61 29 7F AB 88 24 08 B5 30 23 A6 1B  Ê..îa).«ˆ$.µ0#¦.
00000070  C8 9C 10 A0 18 1E AC 9E 62 A1 B6 6A 48 46 9C 60  Èœ. ..¬žb¡¶jHFœ`
00000080  85 87 CB 63 A8 7F 2B D6 30 53 1D 65 7C AC 22 0D  …‡Ëc¨.+Ö0S.e|¬".
00000090  3A 41 E1 64 DC F7 78 E3 F8 DE 04 43 9C 6E EC 5A  :AádÜ÷xãøÞ.CœnìZ
000000A0  D4 A6 D1 7F A9 6C DF 63 DF D5 71 2E C1 D6 5E 55  Ô¦Ñ.©lßcßÕq.ÁÖ^U
000000B0  5F 8E 70 4B B8 53 F2 8D F7 16 2B 3A B9 A1 8A 8B  _ŽpK¸Sò.÷.+:¹¡Š‹
000000C0  9B E0 3E E7 FC 68 CB 8D 9A 95 45 00 E4 A4 EC 15  ›à>çühË.š•E.ä¤ì.
000000D0  F4 1E 80 83 E4 36 81 7A 45 12 00 6C 5A A4 D3 0B  ô.€ƒä6.zE..lZ¤Ó.
000000E0  8B 89 B9 91 39 F7 E2 B7 98 59 39 B3 86 4D 5D AF  ‹‰¹'9÷â·˜Y9³†M]¯
000000F0  58 9E 30 EF CD 2B BC 0C 9F DC E6 C6 9E 71 AA E8  Xž0ïÍ+¼.ŸÜæÆžqªè
```

Incorrect Secret.dat File

To narrow those down and correct them, I went through the file looking for intact strings. I was able to narrow down the incorrect bytes and, because I knew what a few of the strings should have looked like, I was able to correct the incorrect bytes and build the correct key! Manually. #CRYPTOOOO



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  40 50 3A C8 2D 69 F7 D3 02 3A F4 AC 67 DE 4D 3B  @P:È-i÷Ó.:ô¬gÞM;
00000010  81 44 3E 4D ED E1 D5 43 74 56 A8 D2 9F A0 23 29  .D>MíáÕCtV¨ÒŸ #)
00000020  12 6D 0B CE A4 0F C7 F6 BE 9F F7 C6 6D 74 DC BA  .m.Î¤.Çö¾Ÿ÷ÆmtÜº
00000030  D2 D4 0E A3 D5 29 0C 31 F1 08 53 2E B1 0C 79 6A  ÒÔ.£Õ).1ñ.S.±.yj
00000040  29 BC E1 E4 B3 BB 40 BF 69 77 7D 21 55 EA 80 45  )¼áä³»@¿iw}!Uê€E
00000050  7E 43 03 60 93 B4 98 B3 26 8E 35 A1 6C CC 6E D4  ~C.`"´˜³&Ž5¡lÌnÔ
00000060  CA 8F 10 EE 61 29 7F AB 88 24 08 B5 30 23 A6 1B  Ê..îa).«ˆ$.µ0#¦.
00000070  C8 9C 10 A0 18 1E AC 9E 62 A1 B6 6A 48 46 9C 60  Èœ. ..¬žb¡¶jHFœ`
00000080  85 87 CB 63 A8 7F 2B D6 30 53 1D 65 7C AC 22 0D  …‡Ëc¨.+Ö0S.e|¬".
00000090  3A 41 E1 64 DC F7 78 E3 F8 DE 04 43 9C 6E EC 5A  :AádÜ÷xãøÞ.CœnìZ
000000A0  D4 A6 D1 7F A9 6C DF 63 DF D5 71 2E C1 D6 5E 55  Ô¦Ñ.©lßcßÕq.ÁÖ^U
000000B0  5F 8E 70 4B B8 53 F2 8D F7 16 2B 3A B9 A1 75 74  _ŽpK¸Sò.÷.+:¹¡ut
000000C0  9B E0 3E E7 FC 68 CB 8D 9A 95 45 00 E4 A4 EC 15  ›à>çühË.š•E.ä¤ì.
000000D0  F4 1E 80 83 E4 36 81 7A 45 12 00 6C 5A A4 D3 0B  ô.€ƒä6.zE..lZ¤Ó.
000000E0  8B 89 B9 91 39 F7 E2 B7 98 59 39 B3 86 4D 5D AF  ‹‰¹'9÷â·˜Y9³†M]¯
000000F0  58 9E 30 EF CD 2B BC 0C 9F DC E6 C6 9E 71 AA E8  Xž0ïÍ+¼.ŸÜæÆžqªè
```

Correct Secret.dat File :)

I decrypted every file and then extracted everything from file4.gz. File4 contained the key, which we opened up in Notepad++. It says in ASCII block text, "For some reason psifertex really likes Aglets (??? - Not sure about this word really.) In this case it's necessary because the file size should not be a huge giveaway. Though I suppose images would have worked too. Anyway, the key: BuildYourOwnCryptoSoOthersHaveJobSecurity"

And there's your key! Here's all the code that I used to build my secret.dat files. It's a bit messy with the commenting though!

```
import sys
import argparse

parser = argparse.ArgumentParser(description="n.a")
parser.add_argument('--infile1', metavar='i1', nargs='?', type=argparse.FileType('rb'), help=':
parser.add_argument('--infile2', metavar='i2', nargs='?', type=argparse.FileType('rb'), help=':
parser.add_argument('--outfile', metavar='o', nargs='?', type=argparse.FileType('w+'), help='ou

# outfile a+ option opens file for both append and read. New file created for read/write if no

args = parser.parse_args()
temp = open('temp', 'w+')
temp.truncate()
header=open('header','w+')
header.write('\xFF\xD8\xFF\xE0\x00\x10\x4A\x46\x49\x46\x00')
header.seek(0)
key=open('key', 'w+')
key.write('\x40\x50\x3A\xC8\x2D\x69\xF7\xD3\x02\x3A\xF4\xAC')
key.seek(0)

#fuzz.open('fuzz', 'w+')
#fuzz.write('\xff'*256)

#while 1:
#    byte1, byte2 = args.infile1.read(1), args.infile2.read(1)
#    if not byte1 or not byte2:
#        break
#    temp.write(chr(ord(byte1) ^ ord(byte2)))

#temp.flush()
#temp.seek(0)

while 1:
    byte1, byte2 = args.infile1.read(1), '\xff'
    if not byte1 or not byte2:
        break
    args.outfile.write(chr(ord(byte1) ^ ord(byte2)))


sys.stderr.write('Done.\n')
```

Please comment down below if you have an comments or questions for me! Thanks for reading!

--dotKasper

Posted by Delusions Grandeur at 2:00 PM
Labels: CSAW QUALS 2013

No comments:

Post a Comment

Enter your comment...

Comment as: Select profile...

Publish Preview

Newer Post Home Older Post

Ads