Modern Cryptography Lab

Class: CMSC 426

Professor: Dr. Christopher Marron

Group 8:

Austin Furr, Caleb M. McLaren, Colin Seifer

Due: April 6th, 2021, 11:59 pm

Table of Contents

Introduction

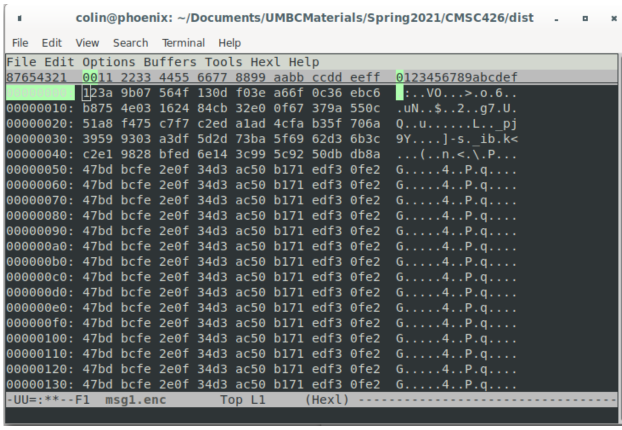
Purpose of this Document:

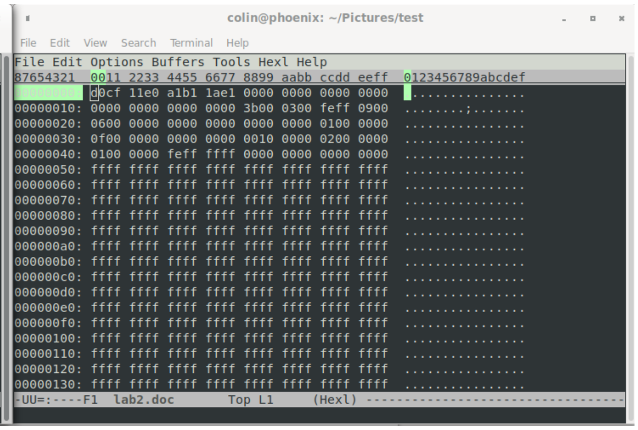
This document must describe the efforts of lab group 8 to analyze the encrypted communications of the fictional Zendian intelligence operatives described in Lab 2. This document must answer the specific questions in the lab description and include materials supporting those answers. This document and supporting materials must be submitted on Blackboard by the due date, April 6th, 2021. This document must bear the signatures of the lab group 8 members prior to submission on or before April 6th, 2021.

Lab Problems

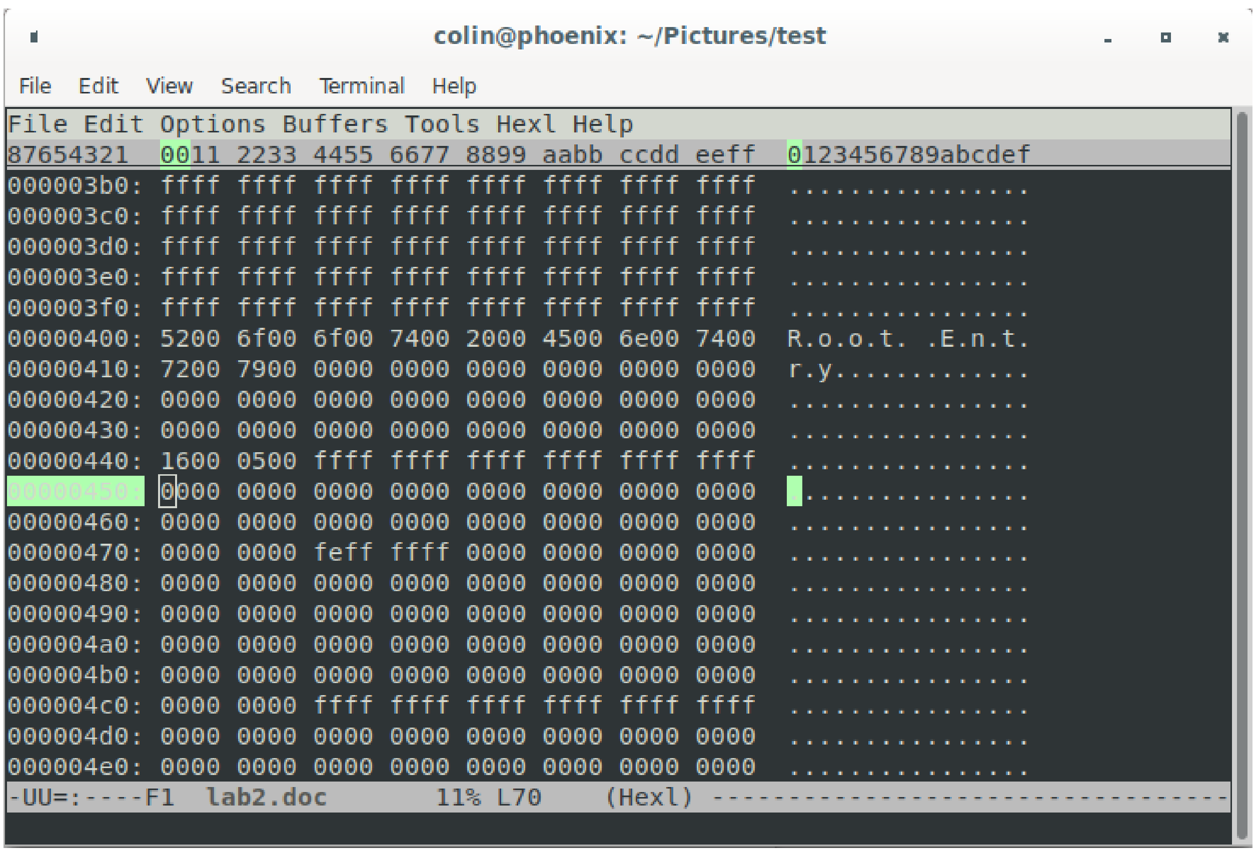
Problem 1

a. The encryption algorithm used on msg1.enc appears to be a block cipher encrypted in electronic codebook mode (ECB). This is likely the case because there are many repeated cipher blocks (as can be seen in the images in part b). The cipher blocks also appear to repeat after 128 bits, suggesting AES encryption is used. If 3DES were used, the cipher blocks would be 64 bits.
b. The most likely format for the underlying plaintext is .doc. Although the file header is encrypted, the length of the file header is the same as that of a .doc file (5 blocks) as shown in the image below.





Additionally, there are two patterns that repeat in the encrypted file. In a .doc file, there are also two repeating patterns when examined in a hex editor: blocks filled with all 0s and blocks filled with all fs (all 1s in binary).



Problem 2

a. The file genkey.py is generating a key based on a length passed into it. It then uses a very large number, N, that doesn't change and a function called time() to generate a "random" number X. After this, a nested for loop is used to go through each byte in an array of the size byte array(key-len). The nested loop multiples X by itself and mods that with N. After that the generated key is shifted to the left by one, X is bitwise, AND with one, and the power is multiplied. This happens for each bit in the byte. This means it happens (8 * key-len) times. To talk more about the value of X, it is generated using the time() and bitwise AND with the value 0x.ffffff, which is the hex value for the color white. As for implementation, there is nothing wrong with the idea, but the number generation of the time() is not truly random. This makes this form of key generation vulnerable to attack.
b. As I said earlier X is being computed using time.time() and bitwise AND with the value 0xffffff. I hinted at it early, but this is a bad generation method for X. The reason for this is that time.time() returns the number of seconds passed since the epoch, which is a fancy way of saying that if two people ran the functions at the exact same time they would get the same value. This is because time.time() is just calling the Unix function gettimeofday(). Since we have the metadata from when the files were created we can use this information and the information gained from genkey.py to get the key used to encrypt each file.
c. Both encrypted messages have very large blocks of repeated hex, this means we can guess the type of file encrypted. ECB was most likely used for this reason. Python Pycrypto library can be used with the decryption for ECB. To find the key we can use functions from the time library and the metadata file that tells us the time the encrypted

files were sent and intercepted. I would recommend the time.mktime(t) function. We should set the time we used to be about a couple of minutes before it was posted and loop up to that time and check all decrypted messages for one that resembles plaintext. Once this is done, we should be able to decrypt both messages that we intercepted.

d. The first thing BIRDMASTER should do is find a different form of generation for the key in genkey.py. There are many different ways to do this, I would start by using a different function than time(). Many modern computers can generate random numbers through mouse movement, noise, or any other outside factors that are not traceable after they have been used. BIRDMASTER could also just use a better algorithm for pseudo-random values. A second thing I would do is use a different form of encryption, ECB is known to repeat ciphertext if there are large amounts of plaintext repeating. This made it easier to determine that the key was not changed during the message being encrypted. Lastly, which may not be possible, BIRDMASTER should try to make his metadata least obvious, maybe encrypt the data and then wait for a period of time before sending the message, this would make anyone watching your activity harder to trace.

Problem 3

a. Message 2 & Message 3

```
To FALLENEAGLE
From BIRDMASTER

Do not worry about money.  There will be nothing if we do not receive the
information you promised soon.  We are making a change to our encryption
software.  It is imperative you switch to the new program as soon as you
receive it.

P.S. Please stop using the old version of the encryption software!  The new
version uses random keys and is much more secure. The old version just used a
hash of the To/From information.

Normal

Default Paragraph Font
```

```
To BIRDMASTER
From FALLENEAGLE

I'm using the new version now.  Would you please explain in detail why the new
version is better?  I will have the documents for you exactly two weeks from
today at 9:00 pm.  Meet at the Japanese lantern in the Tidal Basin.

Normal

Default Paragraph Font

Table Normal
No List
```

a.continued) Code (2 files)

```
# File: antiZendian.py
# Author: Caleb M. McLaren
# Date: April 6th, 2021
# email: mclaren1@umbc.edu
# Description: this Python script must implement an attack on the encrypted files
#   provided in lab 2 of CMSC 426, Spring 2021, at UMBC, Dr. Christophor Marron
presiding.

# Requirements:
   # pycrpyto (pip install pycrypto)
   # genKeyWTimes.py
   # msg2.enc & msg3.enc
   # metadata.txt

# References:
   # https://docs.python.org/3/library/time.html#time.struct_time
   # https://youtu.be/UB2VX4vNUa0
   # https://github.com/the-javapocalypse/Python-File-Encryptor/blob/master/script.py
   # https://www.dlitz.net/software/pycrypto/api/current/
   #

import time as t
from genkey import genkey
from genKeyWTimes import genKeyWTimes
from Crypto.Cipher import AES

# DAY = 3/18/2021
# REF1 = 21:57
# REF2 = 22:02

# mssg2TimeStamp = (2021, 3, 18, 21, 57)
# mssg3TimeStamp = (2021, 3, 18, 22, 2)

# year, month, day, hour, minute, second, day of the week, day count of the year,
daylight savings time
# see https://docs.python.org/3/library/time.html#time.struct_time
startTime = (2021, 3, 18, 21, 56, 0, 3, 77, 1)

#initialize timestamp list
```

```
listMssgCreationTimestamps = [ ]
x = startTime[3]
y = startTime[4]
z = startTime[5]
while x < 22 :
   while y < 63 :
       while z < 60:
           if (y >= 60) :
               x = 22
               y = y - 60
               newTimeStamp = (2021, 3, 18, x, y, z, 3, 77, 1)
               listMssgCreationTimestamps.append( newTimeStamp )
               y = y + 60
           else:
               newTimeStamp = (2021, 3, 18, x, y, z, 3, 77, 1)
               listMssgCreationTimestamps.append( newTimeStamp )
           z += 1
       y += 1
       z = 0
   x += 1

# great got our list of timestamps.
#print(listMssgCreationTimestamps)

listOfFloats = []
for i in range(len(listMssgCreationTimestamps)) :
   #mktime was chosen to convert from local time to seconds since epoch.
   whatIsThisFloat = t.mktime( listMssgCreationTimestamps[i] )
   listOfFloats.append( whatIsThisFloat )

# we have a list of times to replace the returns of time.time() in the genkey.py
#print(listOfFloats)

# write times to file
with open("timesForGenkey.txt", "w") as writer:
   for minute in listOfFloats:
       writer.write(str(minute) + ",\n")
writer.close()

# using list of times to generate keys for those minutes
dictOfKeys = {}
for sec in listOfFloats:
```

```
    newKey = genKeyWTimes(16, sec)
    dictOfKeys[sec] = newKey

# write keys to file for visual comparison
# write keys as strings to .txt file
with open("keysFromTimes2.txt", "w") as writer:
    for i in dictOfKeys:
        writer.write(str(i) + ": " + str(dictOfKeys[i]) + "\n")

# Now to use the AES algorithm in codebook mode with our keys to decrpyt the messages.
#listOfMssg = ["msg2.enc"]
listOfMssg = ["msg3.enc"]
#listOfMssg = ["msg2.enc", "msg3.enc"]

for mssg in listOfMssg:
    with open(mssg, 'rb') as fo:
        cipherText = fo.read()
        for key in dictOfKeys:
            iv = cipherText[:AES.block_size]
            obj = AES.new( dictOfKeys[key], AES.MODE_ECB, iv)
            plainTxt = obj.decrypt(cipherText[AES.block_size:])

            #If key found, plaintext will be obvious in the file preview window of a
Mac file system.
            # for every novel key, generate a novel file to hold the potentially
decrypted text.
            filename = "decr:" + str(key) +".txt"
            f = open(filename, "wb")
            f.write(plainTxt)
fo.close()
f.close()
```

```python
# File: genKeyWTimes.py
# Modified by: Caleb M. McLaren
# Author: ? Dr. Christopohor Marron
# Date: April 6th, 2021
# email: mclaren1@umbc.edu
# Description: this Python function is used to generate keys for the antiZendian.py
script

import time

#modified to receive your choice of time
def genKeyWTimes(keylen, thatInstant):
	n = 27868076781295979681581779653195257l
	x = int(thatInstant) & 0xffffff
	key = bytearray(keylen)
	for i in range(keylen):
		key[i] = 0
		for j in range(8):
			x = (x*x) % n
			key[i] = (key[i] << 1) ^ (x & 1)
	#print(len(key)) # confirm key length
	return(bytes(key))
```

Signatures

Austin Furr Signature: Austin Furr Date: 4/5/21

Caleb M. McLaren Signature: Caleb M. Mclaren Date: 4/6/2021

Colin Seifer Signature: Colin Seifer Date: 4/6/2021

Addendum

This page was left intentionally blank.