Total Encryption of a PNG file using CBC mode 128 bit AES

By: Daniel Palumbo

Abstract

There are many different types and methods of encryption for different datasets. It is important to first understand the datatype in depth before developing an encryption method so that the output is successful in fulfilling needs in the bigger picture. This was learned by performing 128 bit AES Encryption in CBC mode on all bytes of a PNG file. Problems arose when trying to re-render the image essentially outputting a string that is only usable if decryption is available. Upon later investigation into the findings the question was posed "is it even necessary to render an encrypted image?" That really depends on the situation in implementation. The scope of the experiment determines the necessity of the findings.

Introduction

After choosing to perform 128 bit AES encryption on a PNG file, there was much investigating and research to be done. In order to encrypt an image file it was necessary to first find out how the data is actually stored. Java was the programming language of choice since it's very familiar and the AES encryption algorithm was mainly reused from subsequent assignments. That being said there were still many hurdles to overcome while working with a foreign file type. The main goal of this study was to use AES to first encrypt a PNG file to an encrypted hex string and then render the file. If time permitted an add on was to decrypt the file back to the original plaintext. The process implemented was successful in encryption and decryption, however, the main failure was the renderability of the encrypted file. The one thing to note and that was later learned is that there are many different ways to encrypt an image file and after completion it was learned that this implementation consists of the complete encryption of the total file.

Background

PNG

PNG file types or portable network graphics are lossless compression image files that are widely used today. An Advantage of using PNG files is the support for transparent backgrounds creating a sticker -like usable image. The actual contents of PNG files consist of three major categories which are the file header, ancillary chunks and critical chunks. The file header is always the first portion of bytes contained in the file. This makes up the first 8 bytes of information and is a consistent standard throughout all PNG files. The file header essentially tells the renderer that the contents in the file is of type portable network graphic. After the PNG file header is "Chunks," these are broken into two categories labeled either Ancillary or Critical and they contain essentially the rest of the information defining the image. Ancillary chunks are metadata such as the time the image was taken and various information like this. The critical

chunks contain the actual image bytes that are particularly found in the IDAT critical chunks. The Ancillary data functions as optional information that is not needed to render the image, however, the critical chunk data is necessary for display.

Mode of Encryption

Upon further investigation into how image encryption occurs in a real world scenario, the normal mode of encryption pertaining to AES(EBC mode) is not the ideal method. The EBC mode of encryption workflow calls for inputting 16 bytes of plaintext into the AES algorithm and then outputting 16 bytes of encrypted ciphertext. This normally works great on encrypting messages and information that does not contain input that has a high number of repeating bytes. Sadly, image files contain a high number of repeated bytes observed in areas such as having a solid background. Upon encryption of an image using EBC mode of encryption, the resulting rendered image is not the same but some information is still shown describing the file. This is shown below in Figure 1 which is first the original image and the rendered encrypted image. Since EBC mode of encryption is not the optimal solution, CBC mode of encryption is normally used on image files. This is done by first taking the XOR of plaintext by an initialization vector and then if the input file is more than 16 bytes, subsequent blocks of plaintext are then XOR'd with the previous ciphertext blocks. This is to ensure a more random Encryption process and the initialization vector is thought of as a secondary key to create randomization.

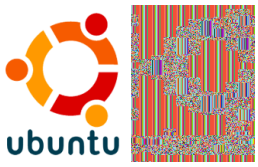

Figure 1.

Toponce, Aaron. *Aaron Toponce*, pthree.org/2012/02/17/ecb-vs-cbc-encryption/.

Experiments

In this implementation of encryption Java was chosen as the language and the encryption method performed was 128 bit AES encryption. CBC mode encryption was implemented outside of the actual AES method. The file type of choice was a PNG image of a beaver found on the internet this can be seen below in Figure 2. The Encryption system consists of three files: the driver file, a png helper file and an AES file. The driver file works by creating a PNGHelper object by passing a path to the image file to be encrypted and also a key. Upon creation of PNGHelper, the constructor method first reads in the image file at the specified path storing it in a BufferedImage object then converts the buffered image to a byte array. PNGHelper then creates an AES object by passing a plaintext key that is 16 bytes of hex string and also a String array containing each individual byte value converted to hex. The PNGHelper object contains a few local members that consist of the buffered image object, output stream for the image and then also the AES object that it instantiates in the constructor method. When the AES object is created, the key that is passed is saved in the local key member and a random initialization vector is generated and

stored locally using the SecureRandom java class. Lastly, a local data array is created to store the hex code of the entire image file which is passed at instantiation and the array is then set as a local member. Once the PNGHelper object is instantiated, the driver file then performs PNGHelper.getEncryption() and then PNGHelper.getDecryption() to successfully output the encrypted hex string that is the image file and output the decrypted image file as the hex string. An attempt was then made to write a new image output file that would be the encrypted file but this was not successful.

Figure 2. “Home.” *Beaver PNG Images Free Download*, pngimg.com/images/animals/beaver.

Discussion

After successfully encrypting the image file shown in figure 2, the output hex ciphertext was then originally converted to bytes and attempted to output a PNG file but the output was not writable. This was not possible since the output does not follow PNG content guidelines. The file writer will not understand the input bytes. After looking into this it made sense as to why it was not renderable as an image since it was quite literally gibberish. The system was able to successfully encrypt and then decrypt an entire image file which is a successful application, however, how important is renderability? If an image file was being sent over the internet it may actually be more beneficial to keep the entire file encrypted rather than rendering the encrypted image file and sending that.

Conclusion

Overall the work encapsulated in the encryption and decryption of a PNG file is rewarding upon success. It was found that different datas can be encrypted in an overwhelming number of ways and it’s important to understand that the solution to a problem really depends on how it fits into your overall project. The scope of this experiment was mainly to encrypt and decrypt an image file and the implementation definitely succeeds although maybe not in an incredibly useful way.

Sources

1.“Portable Network Graphics.” *Wikipedia*, Wikimedia Foundation, 11 May 2021, en.wikipedia.org/wiki/Portable_Network_Graphics.

2.Rosencrance, Linda. “What Is a PNG (Portable Network Graphics)?” *WhatIs.com*, TechTarget, 23 Aug. 2019, whatis.techtarget.com/definition/PNG-Portable-Network-Graphics#:~:text=History%20of%20PNG, t%20need%20at%20patent%20license.

3.*AES Encryption*, asecuritysite.com/subjects/chapter88.