Marist College

Audio Encryption with AES Milestone

Historical Context and AES Implementation in Java for Modern Applications

Tom Mackinson
MSCS 630 Cryptography
Aaron Kippins
May 9th, 2021

Abstract

The goal of this paper is first to provide a historical background on audio encryption, and then to show how I implemented AES encryption in order to mimic how it would be used by modern music streaming applications. Audio encryption and music piracy date back to the early days of the internet, and with the rise of CDs and music sharing applications the need for encryption became more and more necessary. As time went on, measures taken to protect intellectual property grew more and more stringent, and certain companies even took illegal actions in order to stop users from copying CDs. These measures would eventually lead to music streaming as an even more convenient alternative to music piracy, however streaming would lead to the need for encryption, much like how CDs used varying forms of encryption to protect their contents. Finally, I will show how I implemented AES encryption using java as one method of protecting audio content, starting from lab work done in class, then extending those labs to be able to read an mp3 file into java and encrypting the individual bytes, and then taking the encrypted ciphertext and decrypting it back into a playable mp3.

Introduction

Throughout this course I have worked on various types of encryption in the labs. For the final two lab assignments, I implemented AES encryption, both generating keys and the actual steps of encryption with those keys. For this project I decided to use that process and encrypt audio with the same algorithm. In the methodology section I will explain more on how I setup the drivers to encrypt and decrypt the audio files, but the actual encryption and decryption code is based largely on what we accomplished in the lab assignments. The goal of this project was to apply the AES method that we implemented in java and extend it to work on mp3 files. This took quite a bit of research on reading and writing files in java, as well as working with arrays of raw bytes, something I have never done before. Initially, reading an audio file in and converting it to a hex string took too long, however by cutting down the size of the sample file I was able to work on the algorithm itself. With that smaller sample I was then able to implement decryption, and create the finished project that will both encrypt and decrypt audio files.

Background/Related Work

In the early 2000s, P2P music sharing exploded onto the internet thanks to sites such as Napster and Limewire. Music sharing over the internet had a serious impact on the sales of CDs at the time, with studies showing sales of CDs were down anywhere from 14 to 30 percent (Michel: The impact of Digital File Sharing…). With this loss of sales due to illegal file sharing online, the music industry took measures to protect their copyrights and prevent more music from being shared illegally. From taking legal action against Napster in attempts to shutdown the website (Kravets: RIAA sues Napster), to installing security measures on user's computers, even without asking for consent or providing a license agreement (Brown: Sony BMG Rootkit scandal), music companies have gone to great lengths to try and protect their intellectual property, and prevent further piracy from occuring.

The scandal with Sony's BMG rootkit discovered on audio CDs is a perfect example of the steps one company was willing to take in order to curb piracy in the pre-music streaming era. A copy protection measure called XCP was developed by a company called First4Internet, which was used by Sony BMG in order to restrict CDs from being copied. A security expert, Mark Russinovich uncovered that this protection measure made use of a rootkit in order to setup on a user's computer.(Mulligan, Perzanwoski, *The Magnificence of the Disaster*). Rootkits are commonly used in malware in order to hide a virus or other unwanted programs from the user. While in this case the primary goal of the

rootkit was to prevent CD copying, it was installed on user's machines without their consent or approval, and could have potentially been exploited by attackers to further take advantage of their machines.

The use of a rootkit, and the lack of a license or any kind of disclosure about the anti-copying measures go against best practices or ethics for software development. Eventually as the internet continued to grow, music streaming became an ever more popular alternative to purchasing physical CDs. With the rise of music streaming, new forms of security would be needed to prevent another disaster for the industry like Napster. Below I will present an implementation for AES encryption and audio files. By using AES a company can ensure that their property is protected from piracy, while also ensuring that no illegal software is installed on a user's device.

Methodology

Below I will describe how I went about creating the music encryption application. There were quite a few challenges and new parts of java I had to work with in order to actually read in an mp3 file and encrypt it, as well as reading in the ciphertext file and decrypting it. I will go over some of the major steps I had to accomplish and what approach I ended up taking to get the final product working. In addition, I will describe a few of the issues I faced with each step, and why I decided to take the approach I ended up using.

Experiments

To actually go about encrypting an mp3 file, I started by using the code we created in our labs. Our last two labs were to write the AES encryption algorithm and have it encrypt hex pairs using hex keys. This worked fine for the labs, but leads to a few problems when working with audio files. My initial program was designed simply to copy a file. This means read the file in as input, and then output the same file with a different name. I started with this as a first goal since these are the same basic steps my final program would need to perform. The only difference is that I would have to add encryption and decryption in between reading and writing, however I figured that if I could get the actual reading and writing working, the rest would be as simple as connecting my encryption algorithm and writing a decryption algorithm. I also wanted this program to run in the command line. Since the initial idea for this project was to mimic the encryption performed in the back end by streaming apps. This task is supposed to be automated and easily scripted, as such creating a GUI or front end for it seemed to be counter productive, it would take time away from actual research and would go against what this project was trying to simulate.

The first major issue was actually loading the file into java. That was something I've never worked with before, generally any input for a program I've written has simply been a string input. At first I tried to go about reading the file in the way I would normally, with a scanner. Scanner works fine for basic strings or short input streams, but loading an entire non text document this way did not work. My test mp3 file was a roughly 3 minute long song with a file size of about 6MB. Copying this file took a few minutes with a scanner, and lead to a text file that had a slightly smaller file size than the original mp3. This file was unreadable in a text editor, and unplayable by my music player application. I did some more research and quickly saw that scanner was not suitable for this purpose. I left scanner in so the user could enter both a file name and an encryption key, tasks that would be easy to write scripts for, however for actually reading files I had to do some more research. Looking through java documentation and other resources online, I eventually came across input and output streams. These provided me with a way to read a file in and store it as an array of bytes. Converting my program from scanner to input streams was not a difficult task, and by leaving scanner in the project, I could have the

user enter a specific filename to encrypt, which means the program could load any filerequested. Using input and output streams, my initial mp3 file was copied to a new file which played as expected in a music player.

With this first issue out of the way, I then started to look into the actual encryption and decryption process for audio files. Since I was working with a byte array and expected hexadecimal pairs, I decided to work on converting the bytes into hex. Not only did the bytes have to be converted into hex, but they had to be the expected 16 pairs of hex in order to properly work within the algorithm. To do this I decided to take a simple approach and use a for loop, converting bytes to unsigned bytes, and from there converting them to hex strings. This ensured that my input file was broken down into the expected amount of strings, however it is a very slow process. For a single 3 minute song it took several hours to read in and convert, not even counting the encryption time for all of those keys. I decided to cut down the size of the file to a smaller sample, a 15 second clip from the original song. This meant that my computer would be able to read and convert the raw bytes into strings in only a few minutes. This long time to encrypt a song would also not be an issue for a larger streaming company. These large companies would have access to much more powerful hardware then my own computer, meaning that the time it takes me to encrypt a file is much longer than what it would take on the actual hardware that this program would be run on.

After switching to a smaller file, I worked on implementing decryption. Decryption for AES is not something that we worked on in class, however I performed some research into the topic and was able to mostly reuse my initial encryption code. Most functions had to be reversed to work with decryption, but with a little work and some more lookup tables for galois field multiplication, decryption was implemented. One of the first things I noticed with decryption was how much faster it was compared to encryption. Encrypting this same smaller file, a 15 second clip of a song, took only a few seconds, barely longer than the audio file itself. For an application designed to be used in the backend as the main music application is running, this was very promising.

This speed up in decryption I believe is from the conversion process. Rather than converting input of raw bytes into a series of strings of a certain length, after decryption I simply converted the entire hex to bytes and wrote them to an array of bytes. Since the size of this array didn't matter, I could simply store the entire hex output string as a byte array. This array was then put in a file output stream, and stored to a new mp3 file. I believe that the smaller file size helped, however a larger impact on speed is due to the fact that the process of converting strings to bytes, and then writing that to a file is much faster than the process of reading a file and converting it into several arrays of a specific size. This higher speed for decryption means that if this were in use on a user's device, there wouldn't be much of a load time for a song, especially if other techniques like caching favorite songs, or starting to convert the next song on a playlist were also utilized.

From there I simply modified the program to accept a file name as input for encryption and decryption. I left scanner in after the initial failures at reading in a file, however for testing purposes I simply had the program look for files with specific names, such as plaintext.mp3, or ciphertext.mp3. To modify this to allow for a filename to be input, I had to rework some of the driver code, however the implementation of this feature was trivial. With that complete, the music encryption and decryption project was finished.

Discussion

Overall this implementation of AES for audio files is a look into one way in which music streaming applications possibly protect music. With a little historical background on music piracy from the early 2000s, and then the steps taken to ensure CDs were encrypted by their content owners, it's not hard to see why encryption for audio is such an important application. The impact on CD sales from the convenience of services like Limewire and Napster definitely had an changed the way that music companies protect their content. Going off of this historical idea to a more modern application, like music streaming, allowed me to use more modern methods of security, like the AES algorithm that we studied in class, and actually apply it to a real world application. I also got to see first hand some of the pros and cons that come with encryption and the considerations that need to be taken in when working on an application for actual users.

Initially I thought that the long encryption times would be a problem for the rest of the application, however I believe that these longer times are not that big of an issue. First of all, encryption should only need to be performed once when a new song or audio file is uploaded. This should be done on the back end using fairly powerful hardware, possibly a mainframe or a server. My desktop may be modern, but compared to a commercial server being used by a company like Spotify, it is still fairly slow. Because this process is only performed once, and by much faster hardware, the speed I experienced would not be an issue for an actual music streaming company. In addition to the faster hardware, if only encrypted versions of a song are stored, this means that the audio files can be transferred between servers without fear of the original decrypted audio being intercepted somehow. Also, even if the servers were to become compromised, only encrypted audio files would be at risk, further limiting any damage from an attack. The encrypted audio files shouldn't be much larger than the original file itself, meaning that the encrypted file will not take up any more space on a user's device then on a company server. Finally, as I have said before, decryption is the process that would be performed on a user's device, so if that can be performed fairly quickly, then the user experience will be much better. This decryption speed can be augmented by caching a user's favorite song, or by decrypting other songs in a queue or playlist while idle. These techniques would further provide value for the application as a whole.

Works Cited

Brown, Bob. "Sony BMG Rootkit Scandal: 10 Years Later." *CSO Online*, 28 Oct. 2015, www.csoonline.com/article/2998952/sony-bmg-rootkit-scandal-10-years-later.html.

Dowling, Stephen. "Napster Turns 20: How It Changed the Music Industry." *BBC Culture*, BBC, 31 May 2019, www.bbc.com/culture/article/20190531-napster-turns-20-how-it-changed-the-music-industry#:%7E:text=On%201%20June%201999%2C%20a,the%20boardrooms%20of%20record%20companies.

Kravets, David. "Dec. 7, 1999: RIAA Sues Napster." *Wired*, 7 Dec. 2009, www.wired.com/2009/12/1207riaa-sues-napster/#:%7E:text=With%20a%20bankrupt%20Napster%20unable,with%20loans%20totaling%20%2485%20million.

Michel, Norbert J. "The Impact of Digital File Sharing on the Music Industry: An Empirical Analysis." *Topics in Economic Analysis & Policy*, vol. 6, no. 1, 2006. *Crossref*, doi:10.2202/1538-0653.1549.

Mulligan, Deirdre K., and Aaron K. Perzanowski. "The Magnificence of the Disaster: Reconstructing the Sony BMG Rootkit Incident." Berkeley Technology Law Journal, vol. 22, no. 3, 2007, pp. 1157–1232. JSTOR, www.jstor.org/stable/24117456. Accessed 16 May 2021.