AESDroid

Paul Ippolito

Marist College

Dr. Pablo Rivas

MSCS 630L Security Protocols and Algorithms

May 8, 2020

Abstract: In this paper, I propose the idea and methodology of using the Advanced Encryption Standard to encrypt and decrypt messages on an Android application. An issue I face with doing this is how do I go about encrypting or decrypting the user's message accurately without demanding the encryption key to be provided by the user. Can I efficiently and accurately encrypt or decode the user's messages with a random key? How can I use AES to accomplish my mission with this application? This paper delves into my thoughts and ideas and how I have tested them in my Android application I call AESDroid.

1 Introduction: The purpose of this project is to use AES to either encrypt or decrypt a message given from the user. This project is meant to be a semi-simplistic app, at least to the user. The user can either choose to encrypt or decrypt a message (of any length) that they will input themselves. Afterwards, the app will go through the process of the user's choice and return either the encrypted ciphertext or the decrypted message. I will be using AES-128 in order to accomplish these tasks. I am using Android Studio to develop this application, as it is the IDE of choice for developing, debugging, and maintaining Android software. My primary programming language will be Java, as I am already well versed in its usage, syntax, and debugging. However,

my code for encryption and decryption will be written in Kotlin, as I wanted to try out the new-to-me language as a challenge.

1.1 Technology: For this project, I decided to try and challenge myself. I had already developed AES-128 encryption as a lab prior to this project. As a result, this project is a mix of both Java and its newer cousin Kotlin. Kotlin runs in the JVM, but it has vastly different syntax from Java. All the activities are coded in Java, as well as the multitude of error checking methods within them. My AES encryption/decryption is written in Kotlin as to provide a challenge for myself with the new syntax. This app was developed in Android Studio, the typical IDE for Android devices.

2. Methodologies: AESDroid is broken down into three Activities. These are Main, Encrypt, and Decrypt. Main acts as a main menu, where the user chooses to either encrypt or decrypt using menu buttons. The app will then go to the activity corresponding to the button chosen (i.e. Encrypt button goes to Encrypt Activity).

For the Encrypt Activity, the user can put in the message they wish to encrypt and optionally put in a key. The key can either be 16-character plaintext, such as “restinpieces2020” or “coronavirussucks” OR 32-bit hex. If the key is not hex of length 32 or plaintext of length 16 (if they are trying to fill in their own key), the app will not proceed with the encryption process and notify the user. If the key is 16 length plaintext, it will be converted to hex from EncryptActivity.java’s toHex(String s) method. If the user meets all the requirements, the app will then begin the AES-ECB at 128-bits encryption process. It will then return the user’s key in hex and their ciphertext in hex. PlainText can be of any length whatsoever. If it is less than 16, it will be padded with spaces until it is of length 16. If it’s over 16 characters long, it will break it down into Strings of 16 character length.

DecryptActivity.java follows a similar structure to EncryptActivity.java but follows vastly different rules (for obvious reasons). After more research into the topic, even with brute force it would take 1.02 x 10^18 years to crack/decrypt without the proper key. As a result, the key MUST be given by the user and must be in hex. Both the ciphertext message and the key must be 32-bit hex, or else the app will notify the user and not decrypt the message. If the parameters are met, the app will then begin the decryption process, returning the decoded ciphertext in its correct ASCII values in all caps, given the correct key that was used.

3. Experiments: As I have stated, the app can encrypt or decrypt messages of any length by breaking them up into 16 length strings. Naturally, I tested this for both encryption and decryption. I also tested if the auto-generated keys were both valid and usable in the decryption process and yield the desired result. The following cases were very satisfactory.

16 character plaintext with random key

Message: coronavirussucks

Key: 9B76A92BF5C0805675C24A7C79C9B124 (auto-generated by AESDroid)

CipherText: 52F3AD08DA234AF2E0D1BE3D4DE213F0

Key: 9B76A92BF5C0805675C24A7C79C9B124

Decoded text: CORONAVIRUSSUCKS

Less than 16 plaintext with given key

Message: help

Key: restinpieces2020

CipherText: D0164DE38C979DB71CA81D6A7414CC13

Key: 72657374696E706965636573323032303230

Decoded message: HELP

Over 16 character plaintext with randomly generated key

Message: I am really bad at programming, dude

Key: D22CE578D9F5CD18E0FEC9BD2F0B1298

CipherText:

CB6EF8AC629532633FF39EF00B99CF51F76F3B0E08979A66FDC00ACDE4AD4B3729B08

13C6E49D3BF76C8FF54530B59C5

Decoded text: I AM REALLY BAD AT PROGRAMMING, DUDE

4. Conclusion: I claimed this would be simple and efficient. For encryption, the task was not too challenging, even in Kotlin as I had previously done AES-128 in Java. Decryption was a new beast to deal with, and I was deeply disappointed in my inability to decrypt without a key. However, it is at least positive reassurance to the security of AES-128. This app was a fun test of Kotlin-Java integration (I call AEScipher.kt from my Activity.java files) and I think I may use this language more often in the future, should I develop more apps in Android Studio.