The Golang AES File Encryption Project

By: Jordan Murray
Department of Computer Science, Marist College, Poughkeepsie, New York, USA

Abstract - Files are constantly being transported, making them susceptible to data breaches, fraudulent data alterations, as well as other kinds of hacks. Thus, proving the importance of file encryption. The Golang AES File Encryption Project explores the methodology behind file encryption with a Golang server using gorilla mux. This serves a basic html page with a file selector that sends a text file to the backend, using a simple API. Once the file is on the backend, the contents of the file will be encrypted using the Advanced Encryption Standard Algorithm in Golang and the newly encrypted file is returned to the user. A comparison of the AES algorithm runtime in Golang as opposed to java, is also a captivating analysis, due to Golang's growing popularity.

1. Introduction

Data privacy and protection is a very prevalent issue in cybersecurity. Files with sensitive data are transferred every day and are susceptible to many kinds of attacks. This served as motivation and inspiration for researching the Advanced Encryption Standard algorithm and how it can be applied to files.

This work is relevant to work at my internship. We have a basic web application set up as well, where we send files to a back end and store in a database. Thus, learning how to encrypt the files before storing them is useful knowledge as a programmer on this application.

Another motivation for this research is to compare the runtime of Golang on the AES standard, to the runtime of java. This analyzation will provide data on which language is more efficient.

2. Methodology

The Golang Encryption project is currently a Golang server that spins up a basic webpage using gorilla mux. To set this up, I installed Golang using homebrew for MacOS.

```
brew install go
```

I then created a go directory under the prj folder in the repository and added go to my $PATH variable with the following commands:

```
export GOPATH=/Users/jordanmurray/go
export PATH=$PATH:$GOPATH
export PATH=$PATH:$GOPATH/bin
```

After the Golang setup, I installed Gorilla Mux using the commands:

```
go get -u github.com/gorilla/mux
go install github.com/gorilla/mux
```

I then served my custom html page by running the command `go run main.go` because this is the file with the webserver. The html page can be seen by going into a browser and going to localhost:9000. Currently on this page are two text boxes, one for plaintext hex and one for key hex, and a submit button.

Once the form is submitted with values, they are sent back to the Golang server which performs the encryption. The AES algorithm is under its own file, AESCipher.go. The Aes method is called from main.go, where it passes in the values received from the front-end and performs the encryption.

To program the AES algorithm in Golang, I started off with the same technique done in our java lab, with simple plaintext represented in 16-byte hex. This allowed the re-use of the data from the labs as test cases, to ensure the algorithm was working properly and providing the correct output.

3. Analyzing Results

The two commands below ensured that the algorithm and client were running properly:

```
go build main.go AESCipher.go
go run main.go AESCipher.go
```

After running these commands, then you may go to localhost:9000 and test the plaintext encryption. I found that the test cases from the lab 5 earlier in the semester pass the test.

The runtime of the lab 5 test cases on the go algorithm (without sending from the front-end) still has to be recorded. I also have to record the runtime results for the java program as well. Once I have tested most of the labs, only then can I provide a more accurate analysis of which language performed more efficiently.

Simultaneously, when submit is clicked to see the encrypted plaintext, a new page is rendered, and the UI design does not match the original html page. This is also something that I intend to fix, given the time allowed

4. Next Steps

One of the next phases of the project is to test the runtime of the plaintext encryption in Golang vs java more, to provide a more accurate analysis. Another step that I will take to improve upon the algorithm is to refactor the application, so it is fitted for file encryption. The type of file selected to start will be text files.

One technique for file encryption is to read the contents of the file and encrypt the contents as opposed to just encrypting the file name.

To accomplish this, I will programmatically have a scanner that takes in the file from the front end and converts the values into hex bytes. It will then section off the hex into 16 bytes at a time and run it through the original plaintext hex version of the AES algorithm. The program will also generate a random key with the length of 16 bits to also be passed into the algorithm. The key will be applied to all of the lines of the text file, so the file is not encrypted using multiple keys. However, when a different text file is sent, a new key will be generated then, so that not all files are using the same key. This is an attempt to replicate the way that the algorithm would work if using the built-in library that Go provides.

Once all of the contents of the file have gone through the algorithm 16 bytes at a time, the cipher text will be concatenated back together and stored back in the text file. Hence, the new contents of the file are the encrypted, ciphertext version. After the encryption is performed, the encrypted file will be sent back to the client so they can then transfer that file safely. Another option instead of sending the file to the client, the backend could then store the file in a database, like at my internship (not implemented in this project).

To further this research, it would be interesting to learn about the process of decryption using the AES algorithm. To extend upon this application for decryption, the user could send the encrypted text file in hex and the key used to encrypt and have it return the plaintext version of the file. Knowing the key is important for the user if they wanted to decrypt the file. One good test case for this would be to just re-enter the file that was returned to see if the full encryption/decryption process is correct.

Resources:

https://github.com/gorilla/mux#serving-single-page-applications

https://golang.org/doc/