Encryptor - An Application for Secure Storage on Google Drive

Sunil Raiyani, Sahil Agarwal

December 07, 2015

Contents

1 Introduction

1.1 Idea and Motivation

Google Drive is a secure cloud storage and file backup service for photos, videos, files etc. It allows you to access these files from anywhere, using any compatible device. Although Google ensures privacy of the data in the files that are stored on the drive, there is concern about the fact that the data is still not private as far as Google, itself, is concerned.

We have developed a desktop application for Google Drive which will enable users to encrypt their files before storing them on the server. The users can retrieve these files from the drive by downloading them. The application will decrypt the downloaded file and return the original file to the user. Along with encryption/decryption, the application will also check for authenticity of the downloaded file. If it is found that the file has been tampered with, the application will discard all data.

Ensuring privacy of data stored on the cloud, from the cloud service provider itself, is the motivation behind our project.

1.2 Features

The desired functionality here is that:

- It should provide authenticated encryption (and decryption) on a per-file basis
- It should be *easy to use*. The interface should be such that the processes of encryption and authentication are transparent to the user.
- The user should not be prompted to enter a password each time.

1.3 Security Goals

- Authenticated encryption should uphold privacy and integrity of user's data.
- Efficient key management

2 Implementation of cryptographic protocols

We use Authenticated Encryption to provide privacy and integrity. The PyCrypto [3] library is used to implement the cryptographic primitives.

2.1 Initialization: Key generation

A user-given password is hashed using SHA-256 algorithm to generate a 256 bit (32 byte) digest. The hash is stored in the local file system and is used as the key for authenticated encryption. This is done only once when the application runs for the first time.

2.2 Upload: Encryption

Encrypt-then-MAC [2] is the authenticated encryption scheme used. The encryption algorithm is described in Algorithm 1. Below is a summary of the upload procedure:

- Encrypt the file that has to be uploaded using AES block cipher in CTR (random IV) mode.
- Generate a MAC for the encrypted file using HMAC and append it to the file.
- Use the REST API [1] provided by Google to upload file to drive(in a folder named 'Encrypted'). Add an extension (".enc") to distinguish it from non-encrypted files.

Algorithm 1 $\mathcal{E}_{K_e \| K_m}(M)$

$M[1]M[2]...M[m] \leftarrow M$
$C[0] \leftarrow^{\$} \{0,1\}^{128}$
for $i = 1...m$ **do**
    $P[i] \leftarrow E_{K_e}(C[0] + i)$ $//E = AES$
    $C[i] \leftarrow P[i] \oplus M[i]$
$C \leftarrow C[0]C[1]...C[m]$
$T \leftarrow HMAC_{K_m}(C)$
return $C \| T$

2.3 Download: Decryption

The decryption algorithm is described in Algorithm 2. Below is a summary of the download procedure:

- Download encrypted file from drive using the REST API provided by Google.
- Decrypt the downloaded file (if it has been originally encrypted using our application)
- Verify the MAC. If the file is found to be authentic, return the decrypted file to the user.

Algorithm 2 $\mathcal{D}_{K_e \| K_m}(C \| T)$

$C[0]C[1]...C[m] \leftarrow C$
for $i = 1...m$ **do**
    $P[i] \leftarrow E_{K_e}(C[0] + i)$ $//E = AES$
    $M[i] \leftarrow P[i] \oplus C[i]$
$M \leftarrow M[1]M[2]...M[m]$
if $T = HMAC_{K_m}(C)$ **then**
    return M
else
    return $\perp$

2.4 Implementation notes

Since file sizes can be really large, the entire file is not read by the program in one go. The file is read *chunk by chunk* and appropriately encrypted and MAC'd. The relevant Python snippet for this is given in Figure 1. After some empirical tests, we chose 1 KB = 1024 bytes as the chunk size

```
cipher = AES.new(key1, AES.MODE_CTR, counter = ctr)
mac = HMAC.new(key2, digestmod = SHA256)
mac.update(iv)
with open(filename, 'rb') as infile:
        with open(encfilename, 'wb') as outfile:
                outfile.write(iv)
                while True:
                        chunk = infile.read(chunksize)
                        if len(chunk) == 0: break
                        enc_chunk = cipher.encrypt(chunk)
                        mac.update(enc_chunk)
                        outfile.write(enc_chunk)
                outfile.write(mac.digest())
```

Figure 1: Encrypting in chunks

3 User Interface

The application presents a very simple and user-friendly GUI developed using PyQt4 [4] library. Basically, it consists of four dialog boxes that manage the interactions between the user and the application.

When a user starts the application for the first time, it redirects the user to a secure web page. Here, it requests the user to grant read/write access to his/her Google Drive file system. On receiving permission, the application fetches the valid credentials for accessing the user's Google Drive account and stores them in the local file system. Later on, it uses these credentials, stored in the local file system, for interacting with the drive.

In the following sections, we describe the functionality handled each of the four dialog boxes separately.

3.1 Key Generation Dialog Box

Apart from fetching the credentials, the application also needs to generate the key that will be used for authenticated encryption. This functionality is handled in the Key Generation Dialog Box. It asks the user to enter a password(8-16 characters) that will be used as a common password for all the instances of the application associated with the user's Google Drive account. Using this password, it will run the SHA-256 algorithm to generate a 256 bit digest. This digest is stored in the local file system and will be used as the key for authenticated encryption. This is done only once when the application runs for the first time.

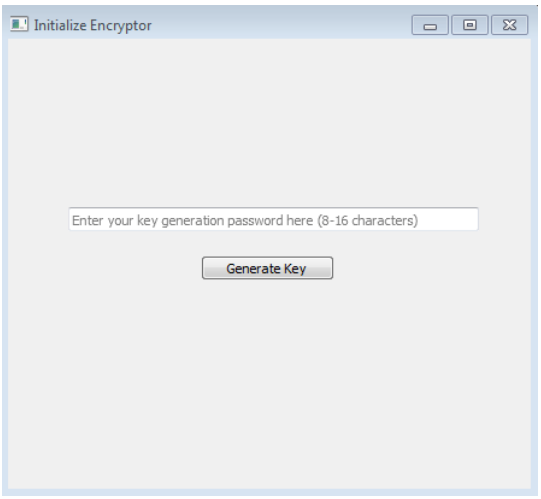

Figure 2: Key Generation Dialog Box

3.2 Main Dialog Box

The Main Dialog Box contains two buttons for performing the two primary functions of the application:

- Upload Button: Pressing this button will flash an Upload File Dialog Box that will assist the user to choose a file from the local file system and upload it to the drive in an encrypted form.
- Download Button: Pressing this button will flash a Download File Dialog Box that will enable the user to download an encrypted file from the drive to the local file system.

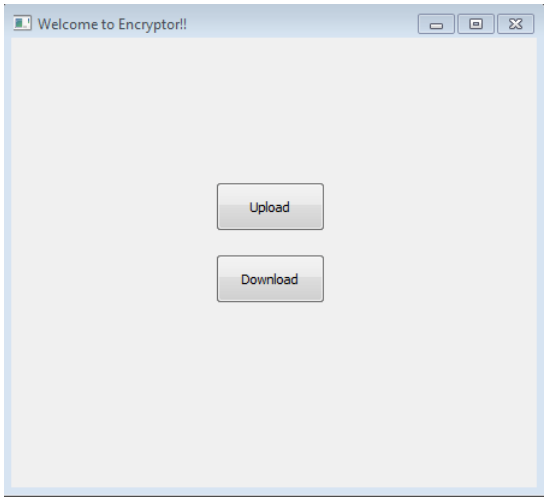

Figure 3: Main Dialog Box

3.3 Upload Dialog Box

The Upload Dialog Box consists of the following two buttons:

- Browse Button: This button enables the user to browse through the local file system and choose a file to upload to the drive.
- File Path TextBox: This textbox displays the path of the file to be uploaded to the drive.
- Encrypted Upload Button: This button allows the user to encrypt the file using an authenticated encryption scheme and upload it to the drive. The encrypted files are uploaded in a folder named 'Encrypted' in the root directory of the drive. When the application starts, it checks if a folder with this name is already present in the drive's root directory, If not, it will create this folder.

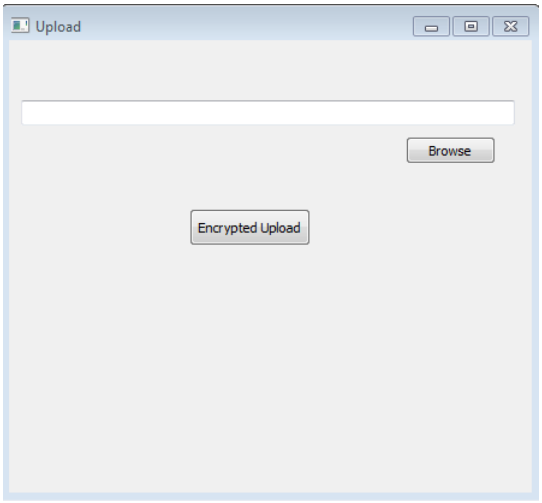

Figure 4: Upload Dialog Box

3.4 Download Dialog Box

The Download Dialog Box consists of a list area, a textbox and two buttons which are used for the following purpose:

- File List Area: This area will display a list of files from the folder named 'Encrypted', which is located in the root directory of the drive.
- Browse Button: This button enables the user to browse through the local file system and choose the directory where the downloaded file should be stored.
- Directory Path TextBox: This textbox displays the path of the directory where the downloaded file will be stored.
- Download Button: This button will download the file, perform decryption and authentication and store it in a folder named 'Encrypted Downloads' in the same directory as the application.

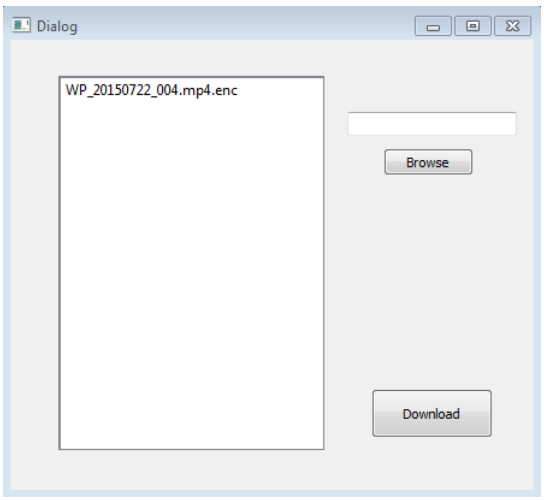

Figure 5: Download Dialog Box

4 Conclusion

We have developed an application that provides privacy and integrity to the user's data stored on the cloud. This application uses an authenticated encryption scheme based on AES and SHA256 to achieve its security goals. Encryption/Decryption is performed efficiently by reading the file in chunks and encypting/decrypting on a chunk by chunk basis. To protect against any tampering attempts, a message authentication code is used which is generated from the encrypted blocks by running the HMAC algorithm.

Future work

- The application can be enhanced further by allowing the user to choose multiple files or folders and upload them to the drive in a single request. Furthermore, the functionality can be extended to enable the user to choose the location on the drive where the selected files should be uploaded.
- The GUI can be made more user friendly by giving drag and drop option to upload files, adding shortcut to the right-click context menu
- Finally, the application could be implemented as a Google Drive plugin

References

[1] API Reference — Drive REST API — Google Developers. `https://developers.google.com/drive/v2/reference/`.

[2] CSE 207: Modern Cryptography. http://cseweb.ucsd.edu/~mihir/cse207/.

[3] PyCrypto - The Python Cryptography Toolkit. https://www.dlitz.net/software/pycrypto/.

[4] Riverbank — Software — PyQt — What is PyQt? https://www.riverbankcomputing.com/software/pyqt/intro/.