Fundamentals of Secure Programming CA3

Higher Diploma In Science In Computing HDCBIBM

Department

Image Encryption in Java

David Harrop (x03298051)

Krzysztof Pawlik (x16138147)

# Background

This project, as defined by CA3 of the Fundamentals of Secure Programming module, poses the challenge of implementing the CIA triangle when designing and developing software.

**Confidentiality**

* + Must only be revealed to authorized users
  + Unauthorized users cannot read information

**Integrity**

* + Must be accurate and complete
  + Unauthorized users cannot alter information

**Availability**

* + Must be reliably available when needed
  + Authorized users can always access information

For our submission, we chose to develop a Java application that can encrypt images. The purpose of image encryption is to prevent unauthorised viewing of the image, whilst allowing an authorised user to still view the image when desired. A real-world example of where this type of application would be useful is in mobile devices. More and more people use their phones as cameras and, according to Irish news sources 15 phones are stolen every day, 7000 since 2015. These images are generally private in nature, sometimes personally embarrassing, yet are very poorly protected. Being able to encrypt these images would protect them in case of loss or theft.

The method of encryption can also ensure that the original image is not altered. Both facets ensure Confidentiality and Integrity of the image.

# Objectives

## Project Problem

Raw images taken by cameras, camera phones and webcams are not protected in any way by default. This makes them vulnerable to exploitation. A famous instance of this was the Celebrity iCloud hack of 2014.

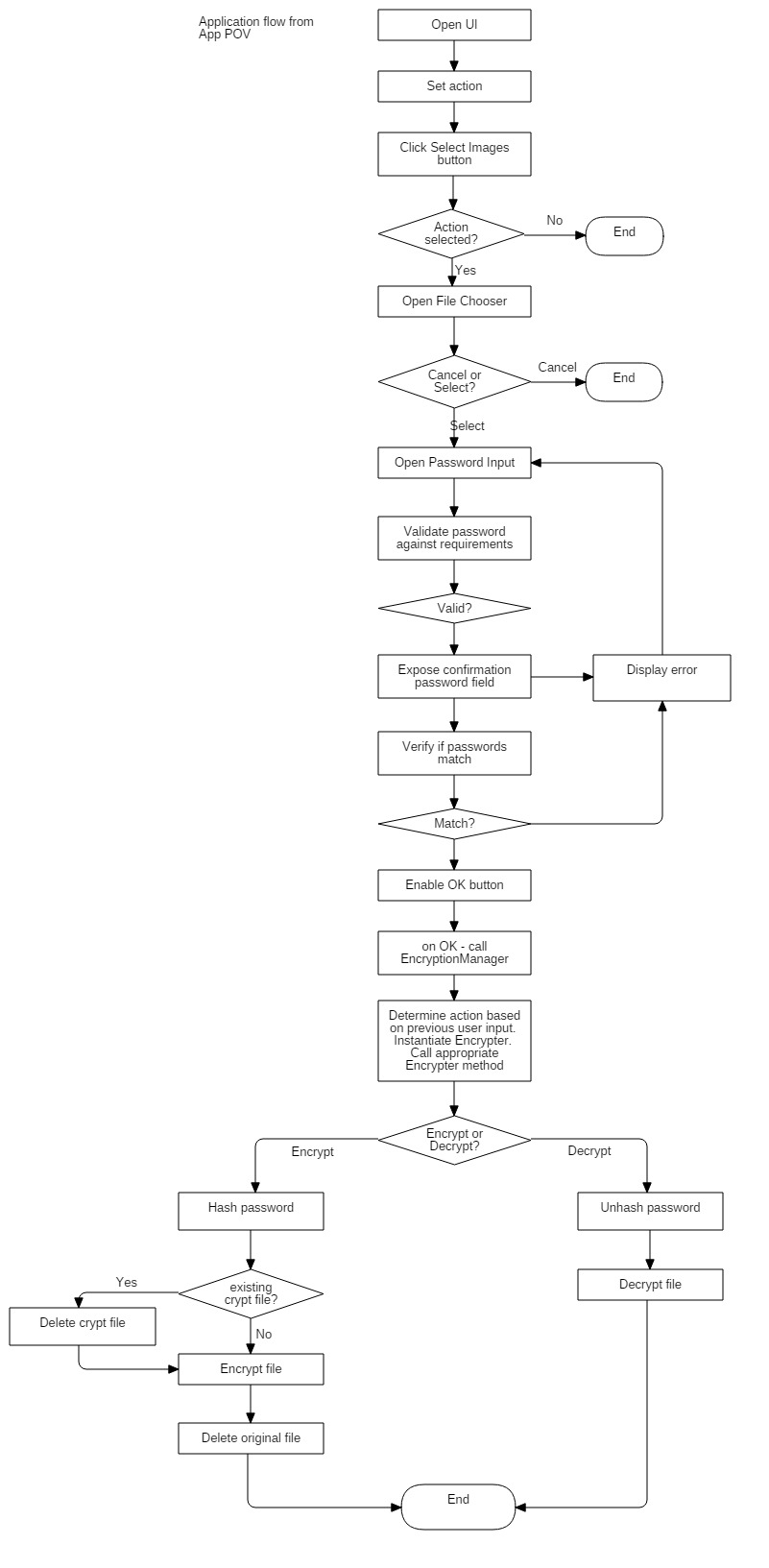
## Project Solution

Using Java, which is platform agnostic, we can develop a simple application that can be used to encrypt images with a password and allow a user to decrypt those same images once they have the password. A password is necessary, otherwise anyone with the application could decrypt your encrypted images, negating the intent behind image encryption.

### Application Execution – User Point of View

|  |  |
| --- | --- |
| Images before encryption |  |
| Run the application and choose to Encrypt.  Then click Select Image |  |
| Browse to the folder containing the image(s) to be encrypted. |  |
| The password must be at least 8 characters long. Until it is, a warning message is displayed. |  |
| Once the password reaches 8 characters it is tested against the minimum password requirements defined by the application. |  |
| Once minimum password requirements are met, the Confirm Password field is enabled. Until the Confirmation Password matches the Original Password, a warning is displayed.  Click OK to begin encryption. |  |
| The hash of the entered password. |  |
| The encrypted images. Note that the original image file has been deleted. |  |
| The result of attempting to open one of the encrypted images. |  |
| Change the extension of an encrypted image to .jpg. |  |
| The result of attempting to open a renamed encrypted image. |  |
| Decrypting the image is the reverse procedure for encryption, so we won’t show the Password input this time. |  |
| The password hash and the file hash being compared. |  |
| The decrypted images. Note that the encrypted images are left behind. If we choose to re-encrypt an image, the original encrypted image will be overwritten. |  |

### Application Execution – Application Point of View



## Objectives

The objective of this project is to apply at least one of the CIA triangle in a Java-based, client application that can protect images by encrypting them with a password.

# System Architecture

We decided to develop a client application without a remote connection because, in our envisaged scenario, the user is choosing to encrypt images on their local machine. Our scenario does not include remote storage of images, hence there is no need for a database in our solution architecture.

Our Java application consists of the following classes:

* **ImageEncryptionUI** – the ‘home’ page of the application, where the user selects the action to be performed. Whilst the brief was the encrypt images, we felt it best to include a decrypt option also. This would allow us to demonstrate the application effectively. This is the first class in our model.
* **FileChooserUI** – we had to provide an interface whereby the user could select the images to be encrypted and encrypted files to be decrypted. This resulted in a second class.
* **PasswordInput** – the method we chose requires a password to both encrypt and decrypt the image. We required an interface that would prompt the user to enter, and confirm, the password. This password must meet some arbitrary rules we established, such as requiring at least one upper case character.
* **EncryptionManager** – once the password is verified the EncryptionManager is called. This class decides which Encrypter method to call, based upon the selection initially made by the user in the ImageEncryptionUI class.
* **Encrypter** – the fifth and final class performs the encryption and decryption of the selected images using the password provided by the user.

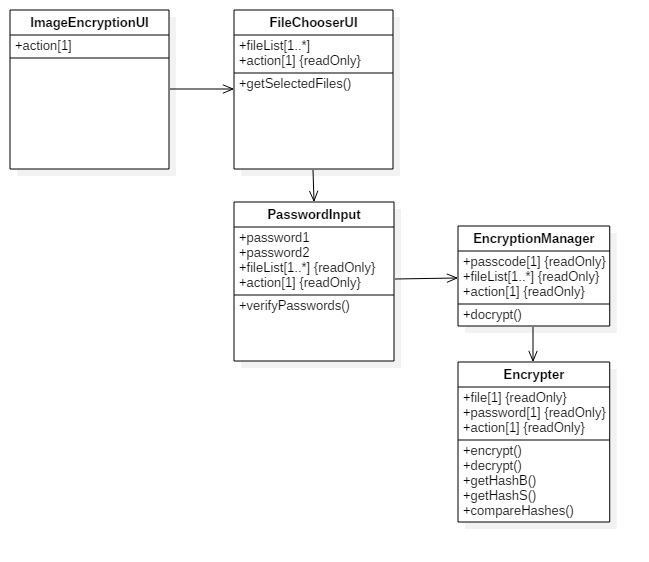


Figure : Class Diagram for the Image Encryption application

# System Implementation

A password must be supplied by the user to both encrypt and decrypt the file. The password must meet the following requirements:

* Be 8 or more characters long.
* Contain at least one uppercase character.
* Contain at least one lowercase character.
* Contain at least one number.
* Contain at least one special character.

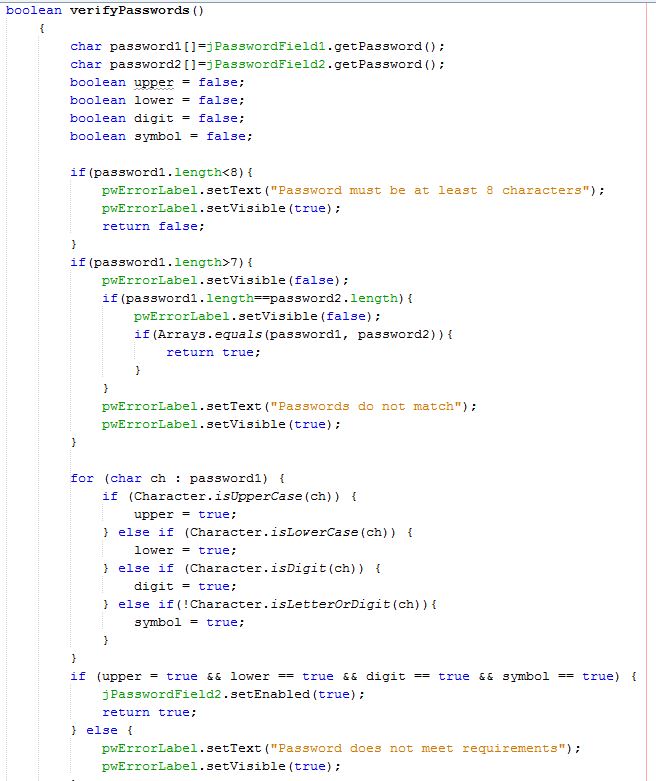


Figure : Snippet of code that verifies password meets minimum requirements

### Steps to Perform Encryption

#### Hash the password

1. Encode the password in bytes using the default charset (key.getBytes())
2. Hash the encoded byte password using the MessageDigest class and the SHA-512 algorithm, creating a 256 bit byte array.
3. Using the StringBuilder class in conjunction with the toString method, cycle through each byte in our array, returning a string representation of each integer with radix 16. Each string is appended by the StringBuilder to generate a 128 character string representing our hashed password.

#### Create the Encrypted Output File

1. Using the FileOutputStream class, create a new output file. The file name is the original input file with ‘.crypt’ appended to it.
2. Write our newly hashed password to the first 128 characters of the file.
3. A byte array of 262144 bytes is created.
4. Using the BufferedInputStream class, read in each byte of the file to be encrypted and convert them to characters, each one is then written to our byte array.
5. The byte array is then appended to our output file.

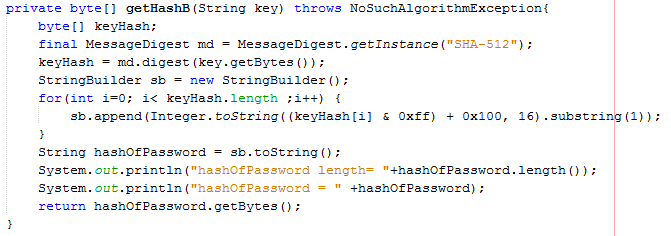


Figure 3: The getHashB method, which hashes the password supplied by the user.



Figure : the encrypt method, which calls the getHashB method of the same class.

When an image is encrypted, the original image file is deleted and an encrypted version of the file is stored in the image’s original location. If an encrypted image file with the same file name exists in that location, that file is overwritten by the encryption process.

During decryption, the first 128 characters of the input file (the encrypted file) are read as the hashed password, which is then compared to the hashed password provided by the user. If they match, the hashed password is then used in the FileOutputStream to create the output file, essentially reversing the encryption method.

We attempted to limit the file selection to just images of type JPG or JPEG, but could not get the fileFilter property of the FileChooser to work successfully.

# Conclusion

Whilst our prototype meets the brief in its simplest terms, to encrypt an image, we determined that this is pointless without a way to reverse the encryption. In fact, we posit that if an encrypted file cannot be decrypted, then the encryption itself was not successful. Therefore in order for us to demonstrate that our encryption was successful, we must also demonstrate the image being decrypted.

As an exercise, it has demonstrated the possibility of encrypting files of any type, but in terms of a practical application there is some way to go before a marketable product can be created. For example, the use of a password to encrypt images works on a low volume basis, but where a user has several hundred images it would be better to use an encryption key. This takes the onus off the user to remember multiple password for different images.

# References

**Article**

[1] http://www.irishmirror.ie/news/irish-news/fifteen-mobile-phones-stolen-every-9939024

**Article**

[1] https://www.irishtimes.com/news/crime-and-law/nearly-3-million-worth-of-phones-stolen-in-last-14-months-1.2572399

**Video**

[1] How to Encrypt and Decrypt an image using JAVA Part1 <https://www.youtube.com/watch?v=xIx_zI1tqAg>

**Video**

[1] Java Projects With Source Code - AES Encryption and Decryption in Java Part – 3 <https://www.youtube.com/watch?v=ICdeI7LzlMI>

Website

[1] iCloud leaks of celebrity photos

<https://en.wikipedia.org/wiki/ICloud_leaks_of_celebrity_photos>