iSESnapchat

Cameron Pelkey
John Reynolds
Mike Brooks

November 6, 2013

1: Introduction

Purpose

This project aimed to produce a secure Snapchat-like client for the Mac OS X operating system.

The project makes extensive use of available system functionality as well as *OpenSSL* [6] for the handling and protection of image data.

1.1 Motivation

The mobile application "Snapchat" has found popularity for its ability to quickly and securely send image and video between users. The novelty of the system lies in the fact users cannot export images or video received and, once viewed, the data persists on the device for only a set length of time after which it is permanently deleted. However a critical flaw in this scheme is that screenshots of received messages may be taken by software imbedded within the mobile device itself, allowing users to easily bypass the security of the application.

Due to restrictions in mobile software capabilities, the Snapchat application is unable to intercept the hardware interrupts that that allow the taking of screenshots, limiting the application's ability to respond to merely notifying the sending-party of this action.

Various software applications currently available on the Mac platform provide the user with a means by which to share media between individuals. However, once this media has been sent through a service the user has absolutely no expectation of control over it. No application presently available provides the functionality of a Snapchat-like application.

2: Description

2.1 Operating Environment

The program is developed on and for the Mac OS X operating system as a stand-alone application.

The choice of working on Mac OS X came primarily as a result of it's availability and ease of development. Most of the group already possessed an extensive prior working-knowledge of development on the OS, while the Unix-like environment provided a necessary level of accessibility to the rest of the group already familiar with such systems.

Additionally, the system comes equipped with the development tools and documentation necessary for properly implementing the system being designed. This lends not only to rapid development cycles but also simplifies the process for building and integrating necessary sub-systems.

2.2 Design and Implementation Constraints

The system is developed with and requires *OpenSSL-1.0.1e* to be installed on the User's system. *Xcode* is required to build the system.

The project itself is implemented in Objective-C. Objective-C is primarily used for creating the necessary graphical user interface and protected windowing system for displaying images. C++ and C code may be integrated into and called from within Objective-C code files with no issue. Of the group, only one member had prior experience with the language. However

2.3 System Outline

High Level Overview

From a high level standpoint, application will:

- Ask the user for a password by which to generate their public and private keypair.
- Present the user with a means of capturing or selecting an image for transfer.

- Allow the user to select a recipient of this image from a list of additional users.
- Encrypt this image for secure transfer using the RSA public key provided by each of those additional users, as well as AES-256 encryption.
- Display the image to the recipient and prevent any attempt at capturing the screen through software.
- Erase all image data once the allowed viewing time has elapsed.

Control Flow

The normal use case for this program may be divided into two distinct operational components: the taking and encryption of an image, and the subsequent decryption and viewing of that image.

Image Capture:

- Open the image capture window.
- On capture, ask for user approval.

  If the image is *accepted*:
  - Allow the user to select a recipients from their contacts list.
  - Encrypt the image using AES-256 encryption, storing that password in a file and encrypting it with the selected recipient's public key.
  - Package both the encrypted password and image with a `.snap` package
  - Store the package to the User's desktop for them to send to the recipient.

  If the image is *rejected*:
  - Allow the User to re-take the image.

Viewing File:

- Allow the user to browse their computer for a `.snap` package to open.

  On *file selection*:
  - Import the file, extracting its contents to a secure location within the application.
  - Generate a random message and encrypt it with the User's public key.
  - Prompt the user for a password
    - If the password is incorrect, delete all files and exit.
    - Else, use this password along with the User's private key to decrypt the AES-256 password.
    - Use this password to decrypt the image data.
    - Delete the encrypted content from the disk.
    - Display the image to the user.

- Start the view timer for the image.

On *timer finish*:

* Delete the image from screen.
* Delete the image contents from memory.

User Interface

The system UI will consist of four distinct components, each within its own display window.

Main capture window

Presented on starting the program. Provides the user with a viewing window from which to capture images.

Contact List / Recipient selection

Displays the private keys of all available recipients to select from.

Received images

Allows the user to select a received `.snap` file from their drive.

Secure viewing

Presents the user with a decrypted image within a secure viewing environment.

3: Development

Working on Mac OS X

The Programming Environment

One of the big decisions the group needed to make initially was what language and environment we wanted to build our project on. For what was already readily available in the language with respect to in-built operating system functionality, Objective C was selected. Additionally, Github would be used to handle a working repository.

One of the large benefits to Objective-C is that, even though not everyone in our group has worked with it, we have all done a mixture of C programming and object oriented programming. Objective-C combines these worlds by extending the C programming language in such a way that makes it an object oriented language [2]. This helps us in our project by allowing us to access lower level details that are offered in C, as well as have the elegance of creating classes and objects that suite our needs as we may do in Java.

Developing on the Mac OS X using Cocoa gives us access to development tools such as XCode and Instruments, which come standard as part of the Mac OS X Developer Tools [2]. These tools will offer us the resources we need in order to implement our project efficiently by allowing us an IDE to work in. Cocoa also offers collections of classes and functionality in packages called frameworks [2]. We hope to be able to use these Cocoa frameworks to implement various functionality that we wish to utilize rather than having to write those components by scratch.

The User Interface

One of the large benefits of the Developer Tools is the Interface Builder that is in XCode. The Interface Builder allows us to create UI components using a tool rather than coding buttons and frames by hand [2]. The UI components in MAC OS X are represented utilizing XIB. XIB is simply a representation of XML [2]. When we create our user interface XCode handles all of the saving of the components in the XIB, allowing us to concentrate more on the design portion.

Screenshots

The fleeting nature of a message sent via Snapchat is what makes the service so attractive. In theory, a message sent is visible only for the duration the sender intends. This behavior can be thwarted on a mobile device. By taking a screenshot of the current contents of the screen, the user would be able to capture a picture of the message (including window "chrome").

The security policy around Snapchat is tailored to the data which Snapchat has the ability to control: the contents of the message. Snapchat only has the ability to control the messages delivered to it, and it has the ability to delete messages. As such, the security policy that Snapchat implements can be expressed as deleting the message from the device.

Obviously, this screenshot-taking action does not break any sort of security policy set forth by Snapchat itself. In fact, Snapchat does not handle screenshots directly. On a mobile device, screenshots are handled by the device's display system, which is married to the operating system. This is to say that the services provided by the device's display system are indistinguishable from the operating system itself, from the app's point of view. To Snapchat's point of view, the message was deleted.

From a meta-policy standpoint, however, the ability to take a screenshot is a fatal flaw in the idea of a message that lasts mere seconds. If the meta-policy is defined as "a user should only be able to see a message for a limited amount of time", and the user may take a screenshot of the message, preserving their ability to access the message beyond the time intended, then the meta-policy is being broken by the system being allowed to take screenshots of the content.

Protecting Against Screenshots on the Mac

On desktop operating systems (and in our case, the Mac in particular), the operating system grants the application more latitude in how it is displayed. The display system is less tightly bound to the operating system, and the display system can be tweaked to suit the application's specific needs.

In researching this project, we came across an interesting property of the Mac OS X built-in DVD player application. Taking a screenshot of the application does not work. Instead of the still-frame you would expect to be in the screenshot, a grey and white checkerboard is used in place. The windowing system seems to be unable to expose the contents of the video to whatever part of the system handles screenshots.

Further research into why the DVD player application does not expose screenshots revealed a property set by the interface that disallows the window system to allow access to the screen buffer of the player-window. By using this window property, we should be able to block access to the contents of our message from other processes, to include the process that takes a screen grab.

Unfortunately, further research proved that incorporating this feature was a highly

involved and infeasible task at the present time. Doing so would require heavy manual handling of the system's GPU and its involvement in rendering the windowing system. There appear to be no other applications, particularly outside of those developed within Apple itself, that make use of this.

Thoughts on Protecting Against Screenshots in Mobile

A few methods could be used to attempt to thwart the use of screen-grabbing a message meant to be temporary. The obvious solution would be for platform-makers (Apple, AOSP, others) to allow for an application to request that the system not be allowed to take screenshots of the active application. Although this solution would be the most globally effective, it would require changes to the platform to work. Alternatively, the screen could be strobed to provide a limited timing window for the attacker to take any screenshot of what would be on the screen – most of the time, a strobed non-message image.

What About a Camera?

Screenshots are only part of the problem. In both cases, desktop and mobile, not only can the user take a screenshot of the message, but they can also take a picture of the physical device (with a message on-screen) with a camera. If taking a screenshot of the screen violated metapolicy, then taking a picture of the screen certainly does too.

There are a number of solutions that could be used to solve this. The first would be to make the contents of the screen obscure such that a camera would have a hard time capturing the image. This could be accomplished by dimming the image or strobing a black overlay on top of the image. These methods rely on the quality of the camera, and the ability of the adversary to time pictures being taken. Both methods are also inherently passive.

Alternately, a more active approach could be taken, where the app on a mobile device or an application on a proper computer could use cameras available to watch for cameras. This method would require computer vision techniques to create an "imaging device identifier". In theory, if the application were forced to run on screens with a camera attached (usually the main screen), then the application could watch for imaging devices pointed at the screen, and blank the screen if such an imaging device were suspected.

4: Final Deliverable

The final deliverable for this project is an implemented, demonstrable Snapchat-like system.

Interface

To maintain simplicity and ease-of-use, the design of the final product was simplified significantly with no loss in the back-end functionality, including security. Minimal user interaction is required throughout the process of taking and viewing images using this system.

Security

In order to fully prevent any and all means by which a user may attempt to take screenshots while the system is running we went with the more practical approach of examining all running file processes on the system and, should one occur relating to an image, deleting that image.

Future Work

We intend to continue to work to improve and extend this project.

First, we would fully integrate the OpenSSL framework into the project so as to avoid reliance on system calls in generating encryption, thereby allowing the program to run on virtually all systems.

Second, we may implement a client-server system by which to automate the sending and receiving of `.snap` files between users. Additionally, such a system would assist in the sharing of contact information between users. With this, security would be entirely out of the hands of the user.

Finally, we would approach a safer way of viewing images that did not require such an extreme measure as was necessitated at this point. Ideally, we would interrupt all processes attempting to save shots of the screen using integrated hardware capabilities.

Bibliography

[1] Apple. *NSWindow Class Reference*. 2013.

[2] A. Hillegass and A. Preble. *Cocoa Programming for Mac OS X*. 4th. Upper Saddle River, New Jersey: Pearson Education Inc, 2011.

[3] Itseez. *OpenCV (Open Source Computer Vision)*. Software. 2013. URL: http://opencv.org/.

[4] Kitware. *CMake*. Software. 2013. URL: http://www.cmake.org/.

[5] Stephen Lombardo. *openssl-xcode*. Software. 2013. URL: https://github.com/sqlcipher/openssl-xcode/.

[6] The OpenSSL Project. *OpenSSL*. Software. 2013. URL: http://www.openssl.org/.