Progressive Authentication in iOS

Genghis Chau, Denis Plotnikov, Edwin Zhang

December 12th, 2014

1 Overview

In today's increasingly mobile-centric world, more people are beginning to use their smartphones for important tasks. This means that many applications will need increased security in order to prevent any unauthorized access. Especially since applications are often always logged in, a careless user who leaves his or her phone unlocked is vulnerable to having potentially malicious users easily access sensitive information in apps like mailing or banking applications. In order to counter this on the iOS platform, we build a framework that allows developers to use per-app authentication. We tackle this problem from the developer's perspective, as OS-level modifications are much more difficult due to iOS source code being unavailable for analysis. Along with the framework, we complement the system with additional authentication methods, including a server that allows for using one-time passcodes to log into applications. Finally, we discuss further extensions to our work, and their security implications.

2 Background

2.1 Design

Our contribution is providing an iOS framework for developers to use in order to force users to reauthenticate when they browse away from the application. This prevents malicious attackers from having access to all applications if they are able to get past the phone lock. Currently, a number of authentication schemes are available for use in our framework, including an alphanumeric password, TouchID, Android-style lock patterns, and one-time passcodes. We also provide an Django server that will allow users to generate the one-time passcodes to log into the application. This gives users the ability to use passcodes resistant to observation, as well as allow authorized users (i.e. friends) to access the application without having to reveal his or her real password.

2.2 Threat Model

Our system operates under the assumption that the primary attacker will not be able to compromise the iOS operating system, and cannot tamper with the hardware. However, the

reality is that some attackers may have these capabilities, and we address how our system can be modified to protect against such attackers in potential further work. Our system does directly address attackers who can gain possession of a device at any time, and can employ any other tactic aside from the ones addressed above.

We also provide a web server that allows for the generation of time-based one-time passwords. We analyze the security of this server by looking at the potential damage an attacker can do, assuming he or she can compromise the server and access any information stored on it.

2.3 Code & Notes

The code for our iOS application (which includes an example application that employs our framework) can be found on Github, and the code for our Django one-time passcode server can also be found in our repository. (Note that "Github" and "our repository" are links.)

We note that due to an inability to acquire an iOS Developer's License, the iOS code has been tested exclusively on an emulator. While this allowed us to test most of the functionality, some of the hardware-based features (for example, the TouchID authentication method), may need further testing. We also note that the Django server should be deployed with an appropriate proxy and signed certificates to run on HTTPS. This prevents potential man-in-the-middle attacks and secures data transmission, but since this functionality lies outside of the capabilities of Django, we do not address it in our code. A security policy like one described in ForceHTTPS would also be beneficial.

3 Implementation

3.1 iOS Framework

We implemented a framework to allow app developers to import the framework into their app and set up progressive authentication on the app. The types of authentication the framework supports are passwords, TouchID (although it is not tested, due to not having a Developer's License), pattern (similar to the Android unlock), and the one-time passcode. After authentication is set, whenever the app opens, the user will be forced to authenticate themselves. Too many incorrect guesses will close the app. Each subsequent incorrect guess will close the app until the user enters the correct password. Whenever the user wants to change their password, they first need to authenticate themselves by entering their old password again.

We also implemented a very basic app to demonstrate the integration with our framework. The app prompts the user to select what kind of authentication they want, and then calls the appropriate view controller to create the password and then shows them an authenticated view, where the user has the ability to turning off authentication and resetting their password. To change authentication methods, the user needs to delete authentication, restart

the app and then set the authentication method on app launch. However, this functionality is completely on the app side and not related to our framework.

To integrate the framework, the developer needs to import "ProgressiveAuthentication.h" and call [ProgressiveAuthentication sharedInstance] to initialize it. After that, they specify what kind of authentications they want and then load the appropriate view controller to create the password based on the authentication type. They do not, however, need to put in code to launch the authentication view when the app opens, as the framework does that automatically.

3.2 One-Time Passcodes

We also give users the ability to generate and use one-time passcodes, which provides users with the option of having dynamic passwords at the cost of having a trusted server. Since users will need to register an account with the server, the security of this scheme relies on the strength of the password chosen on account creation. With the advent of password managers, we advise users to utilize long randomly generated passcodes (to avoid easily-guessable codes) and password managers to remember them.

After a user registers with our Django server, they can register any number of applications to use one-time passcodes, given that the applications utilize our iOS framework. The application and server do not need to communicate with one another aside from a synchronization action at application registration. When a user registers the application, the server randomly generates a secret key (using Python's `urandom` function) and encodes it in a QR code, which the user scans using the application. (Alternatively, we also provide an optional API to get a new key from the app itself.) This provides a shared secret key between the application and the server. The QR code is given a random file name on the server, and is deleted on the first GET request to it. This will reasonably prevent attackers from trying to guess QR code image names and extracting secret keys - if an attacker somehow manages to guess a file name before the user loads the image, the user will not see the QR code (as it has been deleted). While our version of the server does not currently deal with potential race conditions when the attacker and user access the QR code nearly simultaneously, we expect this case to be rare, and can also employ file locking to prevent this.

After this initial synchronization step, the secret key is stored in a server database, but encrypted using AES with the user's password, which the user must enter whenever an application is registered, or a new disposable key is generated. Again, we expect password managers to improve the user experience with this scheme, and this prevents any plaintext sensitive data to be stored persistently on the server. Since passwords are hashed with salts and app-level secret keys are encrypted using the user's password, an attacker who compromises the server and looks through the databases should not be able to figure out users' passwords nor secret keys.

The actual disposable passcode generation is done by using time-based one-time passcode generation algorithms. Here, we use HMAC-SHA512 to generate a code using the Unix time

(divided by 60, to represent time steps of a minute) and the secret key (decrypted with the user password, which the user must enter). We represent the result from the HMAC algorithm in hexadecimal, take the last byte as an offset, and use the next 6 letters as the disposable passcode. Users can then use this passcode in the application to authenticate themselves. This is similar to the algorithm described in RFC 4226[1]. This algorithm is also replicated in the application framework, allowing the application to generate the same disposable key to verify user input.

From our use of Django, our web server is safe against vulnerabilities such as XSS and CSRF, preventing attackers from injecting malicious code or tricking users into sending undesired requests. We also employed password hashing and AES encryption as described earlier, so that no plaintext secrets are stored persistently and attackers who have compromised the server cannot learn anything from databases (although they could still wait for users to enter in their passwords manually and steal them then). We also chose to use HMAC as it allows us to generate secure passcodes given that the attacker is unable to determine the secret key.

4 Conclusion

Because iOS does not support operating-level modifications to improve application security, we created a framework for developers to use that would address the problem. By giving developers an interface to implement progressive authentication, we hope that more sensitive applications will employ it and provide the additional layer of security for its users. Creating a server to issue disposable passcodes should also allow security-focused users to better secure access to their applications, as well as allow friends or family to access applications temporarily without needing to reveal the main passcode. We also provide the foundation for further improvements in iOS application security, described in the next section.

4.1 Further Work

An important extension that we can use progressive authentication for is encryption of data using a user's password. In order to prevent against any attackers that might have compromised iOS (for example, by jailbreaking the phone), applications can use the user's password to encrypt any data that is stored persistently, as well as any data stored in iCloud. This also prevents attackers who have compromised a user's iCloud account from extracting all information stored there - any attacker who tries to do this would be presented with the authentication screen, as the iCloud data is encrypted.

Additionally, the developer still has to do a some work in terms of figuring out what kind of view controller is needed for password creation. To fix this, we could create a global function that only needs the authentication type and launches the appropriate view controller to create the password, instead of forcing the developer to do this. However, we did not have time to implement this.

[1][RFC4226] M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., and O. Ranen, "HOTP: An HMAC-Based One-Time Password Algorithm", RFC 4226, December 2005.

In addition, our framework currently stores some sensitive information (for example, token generation secret keys) on the phone, which is not ideal (as an attacker can look at persistent data). To help with this, we can instead store this information using iCloud's Keychain, which is designed to store sensitive information. While we were not able to implement this in this version of the framework, modifications should not be difficult. Additionally, the secret key used for the one-time passcode is not encrypted when it is stored on the device, so if a user were to figure out the secret key, it would be easy for them to generate valid one-time passcodes once they figure out our algorithm. This could be fixed by encrypting the secret key when storing it and then decrypting the key upon retrieval.

Other additional improvements to the framework that we planned to implement, but were not able to do so in time, include revoking secret keys (which could be done by giving an authenticated user the ability to delete the secret key), and allowing for user profiles, to allow multiple users to use differing secret keys on the same application.