Pretty Good Chat

Alex Grinman
agrinman@mit.edu

Philippe Schiff
pschiff@mit.edu

Jonathan Stoller
ystoller@mit.edu

1 Abstract

We present "Pretty Good Chat" (PGC), an iOS messaging app that lets users communicate securely without requiring *any* trust in the server and with an increased level of identity privacy, while still maintaining simple usability. The PGC app uses asymmetric key encyption, and provides a very simple interface for users to exchange public keys in the app using QR codes. We also provide a brief analysis of *Telegram* and *TextSecure*, two of the most popular "secure" messaging apps, and a description of future work.

2 Motivation

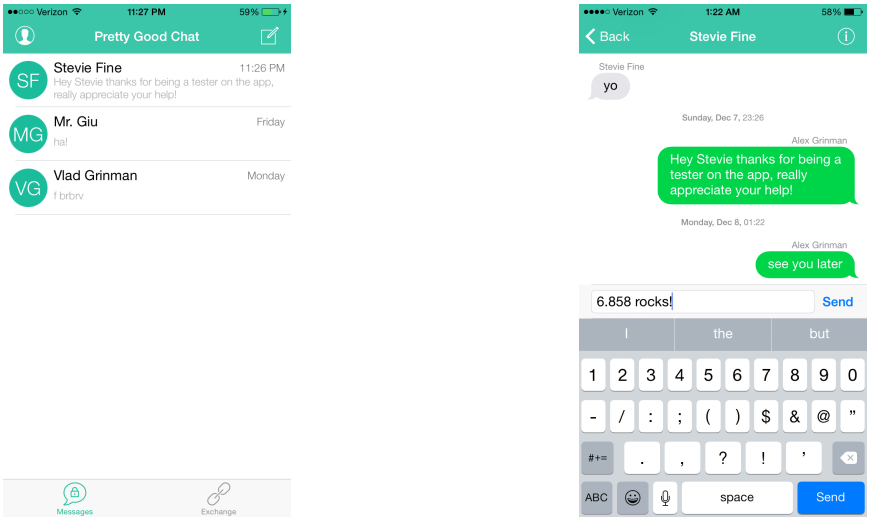
Over the past few years, chat apps have exploded in popularity on the mobile platform. Recently, due to frequently recurring news stories about data leaks and hacks, some companies have created what they claim to be "secure" messaging apps. By the very nature of an internet messaging system, all chat messages must pass through some central server that all chat clients must subscribe to. However, herein lies the problem. The messaging server is a mysterious black box, whose security properties are difficult to ascertain. In fact, in the current landscape of mobile apps, most backends are hosted on third-party services such as Amazon's AWS, Heroku, Redhat's OpenShift, or some other Virtual Private Server (VPS). Hence, the server resides in an unknown location that is assumed to always be trusted by app developers. In addition to trusting all the employees that work for the app company, users must also implicitly trust the services that the app is in contact with to not abuse their promise of security.

Our concern is that too much trust is being placed in the individuals and services that manage the service and data. The attack vector that we are worried about is when an adversary has full control of the messaging server. We can imagine a scenario in which a chat app company hires an employee who is an adversary and is able to change essentially anything and everything in the server application.

Many of these chat apps claim to be secure by having clients encrypt messages using the Diffie-Hellman Key Exchange protocol (or something similar) to establish a symmetric key between the two clients. The server, by definition, is a Man-In-The-Middle (MITM) in the communication path between the clients during the key negotiation protocol. Thus, if the server is completely controlled by an adversary, the protocol can be compromised and the server can decrypt all messages sent by the clients, in a way which is undetectable to the clients unless they compare their symmetric keys through a side channel.

3 The PGC App

In this section we will describe the PGC app design choices including public key exchange, key management system, storing the private key securely, revoking the public key, and data syncing across all iOS devices, and finally possible attacks on the app. We will also describe the structure of the server and how it stores messages.



(a) Conversations UI (b) Messaging UI

Figure 1: App UI

3.1 Visual App Design

Figure 1 above shows the home screen of the app where the conversations are displayed, and the messaging UI, which closely resembles the default text messaging UI in iOS.

3.2 First run

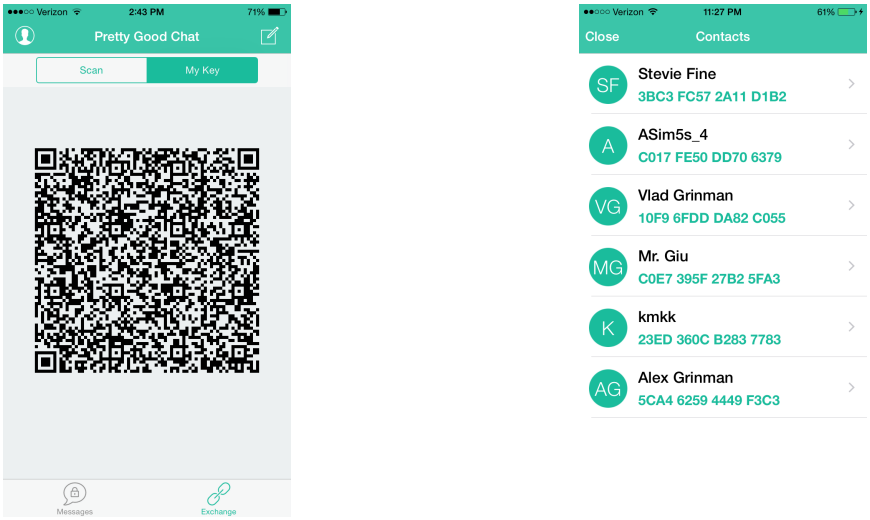
When a user first installs and opens PGC, the app generates a public/private key pair and asks the user for a username. The username is used only for in-app purposes and is never used to identify users. The identifying information sent to the server is a hash of the public key, known as the public key *fingerprint*, which we consider to be a small amount of information about the user. Therefore, the user's identity is private to the chat server.

3.3 Exchanging Public Keys

To exchange public keys, two users meet in person and they both open the app. As shown in **Figure 2(a)**, the PGC app generates a QR code representing the public-key of the user. Using the camera in the app, both users scan each other's public keys, after which the new contact information is stored on the device. The contact information consists only of the username and the public-key. The app maintains a "phonebook" of public key to username mappings. Now the user can open the contact list, as shown in **Figure 2(b)**, and start messaging the other user. Note that users only have to meet in person once, unless they revoke their private keys (as explained in the next section).

3.4 Messaging

A user selects a friend to message from the contact list, as shown in Figure 2(b), and begins a conversation. The plain text message object is encrypted with the recipient's public key, and the message is padded with random bits. Each plain text message object also contains a date. The encrypted message is signed with the senders private key. Finally this payload that contains the sender's public key fingerprint, the recipient's public key fingerprint, the encrypted message, and



(a) Conversations UI

(b) Messaging UI

Figure 2: Key Exchange QR and Contact List

the signature is sent to the messaging server. The server's storage of this payload is described in section 3.8.

3.5 Key Management, Syncing Data with iCloud, Storing Messages

The private key is stored in the iOS keychain which is protected by both encryption and sandboxing. Each app has its own keychain sandbox that other apps are prevented from accessing. Additionally the keychain is encrypted using the user's phone passcode. It is important to note that all new iOS devices use TouchID which is an implicit passcode that the user cannot turn off, and must setup before using the device. The keychain cannot be read at all without unlocking the device, and this thus prevents both other apps on the device and outside adversaries from stealing the private key. With iCloud Keychain turned on, the keychain is securely synced across all the devices that the user enables for this feature. Therefore, if a phone is lost, stolen, or broken, the new device that a user signs into with their iCloud credentials will be able to sync the private key (which is stored in encrypted form, on a sandboxed iCloud storage space) and resume normal use.

The "phonebook" of public keys is also securely stored in the keychain in encrypted format, also implicitly in the app sandbox. This means that outside sources or other third-party apps will not be able to modify or even read the user's public key contact list.

To improve efficiency, messages are stored on iCloud, but they are encrypted with the user's private key in order to make them unreadable to an outside user or even an Apple iCloud team employee.

Therefore, we have both security but also usability since even losing a phone doesn't mean that the user will have to re-exchange keys with all of their contacts.

3.6 Denial of Service Attack on PGC

We found only one plausible attack that PGC is vulnerable to: a denial of service attack in the form of dropping messages. Since our adversarial model is the actual messaging server, it is very easy for

the server, at any random point, to drop a message and not store it in the database. While we argue that the server cannot target which messages to drop, due to them all looking identical and being encrypted, the server still may target a specific public key fingerprint to deny messages to. This is not a very meaningful or strong attack, but it is something that is inherently possible in any system where the server is completely controlled by an adversary for some amount of time.

3.7 Revoking Private Key and Generating New Keys

We provide a simple feature that will allow a user to revoke their private key if they fear it was somehow stolen or exposed. This feature sends a "revoking message" to the entire contact list of the user, which has a special parameter indicating that the contact should remove the user from their contact list. The clever part of this feature is that even if the server was compromised (independently of the private key being stolen), the server would still not be able to distinguish this "revoking message" from any other message, because the payload sent to the server looks identical to a regular message. Therefore the server could not drop this revoking message without dropping all messages. After revoking messages are sent, the app will generate a new public/private key pair, replacing the old one.

3.8 Server Storage Data

The server stores only the *Message* object, which has the following properties. An 'id' for the row, a 'to' field representing the public key fingerprint of the recipient, a 'from' field representing the public key fingerprint of the sender, a message field containing the encrypted message (plain text or revocation signal) with the recipient's public key, a signature field containing the signature of the encrypted message field with the sender's private key, and a digest field representing a SHA-256 hash of the *encrypted* message. An example row is shown in **Figure 3** below.

id	sender	recipient	message	signature	digest
35	5ca462594449f3c3	7333f095b1f21eeb	XfD9XpRxF9tWtULJ6dpjw8OfNBQ8n w3px9bsyGlxMgvI8RKcYYlP/zbW+/ uF8JXC CvMfRht5RuSihAdX2Gr34x7CFd4GQ +YwHAVJKBjTfaoO6aJ2kePQk2yDrr k1tZX9 wI3o4nHD1kOqeiK32+IdDyVdy+74u HYi/NLMsWSeWe28c3EaEGrd1dhqwo ZgCXpC BEIOnVtssnlC+s+Xnqv99X4rpsBr8 OYQGtEociCY3Ehj1ONX9pglwelcl7 7B3zp/ cW1PbtaW4Grjb1mDru3UbzTYO4zm6 Iz1IlKANwKSDXadx1buwYdX70NjCI lxveNt 7hiD1rsvdU8vZB4loSeglA==	a65RzlYcoGZFA5yZJu2N/mG7vy2J8 rMu2U+79NsxTDoYYI7cIv+GooXglD gWuHrQ t57zKkfQ+PT/MgDToj6lOuK1n1n5B COuvbkc/kRnHXY7yKvDWUn0hxFdvJ ubxJFA Mt+suKACj1TFeuod+lNeDsEc2oVg4 FAJEAsiASqeci44N39IVU+iyeEK7j VLpIBL NRpywlqdBkTNUkJ0bL/DrtU5wcssv qt6+mf1HkkQGnjSkiQ3q1Rg8erPnV 9JWTRd b3jpWVNqBARPtOBqP2uVMbuond3r6 9sweXqI7jYXMovUbtM+bndETZ/coQ sBbfmp Ul1vxMUiwS4jHfxp9sDJ3g==	7PD5lAE84WCQXaDw+GQkXeIDP8qwx gBzoPnAIs+A56E=

Figure 3: Sample Message Database Entry

4 Analysis of Telegram and TextSecure compared to PGC

In this section we compare PGC to two of the most popular "secure" chat applications.

4.1 Telegram

Telegram is a free messaging service similar to Whatsapp, that allows users to store content on the cloud and sync information across multiple devices.
As far as security is concerned, users that do not trust the server (i.e. Telegram) are provided with a method of communication that provides an excellent security guarantee without any dependence on the server. This method of communication is called "Secret Chat", and is implemented as follows:

1. Let Bob (B) want to communicate with Alice (A). The end goal is for A and B to maintain a secure channel in which information is encrypted using the Diffie-Hellman protocol.

2. All information exchanged between A and B is routed through the server. The communication channels between the server and the users rely on the use of Diffie-Hellman as well. In order to prevent a MITM attack on these channels, the initial key exchange is encrypted using the public key of the server, which comes installed in the app. (The App Store / Google is assumed to be secure, an assumption PGC makes as well.)

3. As stated above, the DH key exchange between A and B is also done via the server. Since information sent to and from the server is encypted asymmetrically using the server's public key, we aren't worried about third party adversaries. However, the server itself is in an excellent position to launch a MITM attack between A and B, if it should desire to do so.
   In order to prevent this from happening, users that set up a "Secret Chat" are each presented with an "identicon" (a pixelated image). If the identicons are identical, the users can rest assured that no MITM tampering was done by the server.

At first glance, Telegram seems to be an extremely secure alternative to PGC. However, after digging slightly deeper, there remain three strong reasons to prefer PGC over Telegram:

1. Although the "Secret Chat" key exchange can indeed be done reliably, since the keys are stored locally on the phone, every time the user switches or formats his phone a new key exchange will need to be done. Needless to say, this is a considerable inconvenience for the user, who will have to create new keys and examine the resulting identicon for every "Secret Chat" that they previously had.
   In PGC, keys are stored on the iOS Keychain as described earlier, which enables the user to use the same key indefinitely, regardless of whether or not the phone is switched/formatted.

2. Telegram also requires knowing the contact information of the user, while PGC allows users to maintain complete anonymity (our app only requires a username which the server *never* receives).

3. The final issue worth mentioning is primarily one of convenience. In PGC, the key exchange doesn't require any significant user involvement (asides from a handy in app QR code scanner for users to scan each other's public keys). However, Telegram requires the user to manually compare two images. If there is a subtle difference between the two images and the user fails to observe the difference, a MITM attack may succeed. This risk does not exist in the context of PGC.

4.2 TextSecure

TextSecure is an open source encrypted message application developed by OpenWhisper Systems which claims to provide end-to-end encryption of messages. Both SMS messages and chats sent over data networks can be encrypted using TextSecure. It also allows for interleaving of "secure" messages with regular ones, where the "secure" ones have a lock icon suggesting that they were sent securely using TextSecure. Group chats are also possible. Broadly, TextSecure's protocol works as follows:

1. Let Bob (B) want to communicate with Alice (A). B will initiate by contacting the TextSecure Server S. First B registers his phone number, along with a few preferences, with S. B then generates 100 "prekeys" which he then sends to S. These prekeys are signed key exchange messages generated by B before any communication with anybody. Whenever a source wants to initialize a communication with a destination, the source requests the destination's next

prekey from the server. This way, the destination does not have to be online for the key exchange to be performed. A goes through this registration as well.

2. B then requests A's next prekey. It uses this prekey, along with A's long term public key to perform a DH key exchange. B uses this with his own key exchange half to generate a secret key. Then he encrypts his message with this secret key. B then sends the encrypted message to the S, who upon receipt does some verification and sends it on to A.

3. When A receives the message she can choose whether to respond. She now has all the necessary information on her end to calculate the key exchange and communicate with B.

In addition, in order to allow verification of public keys, TextSecure, like Telegram, allows users to generate QR codes representing their public keys and allows them to compare them.

While TextSecure seems to have a good security model there are a number of issues not addressed in TextSecure which would give a user a good reason to use PGC instead:

1. The server is relied upon for storing the prekeys and public keys of each of its users. This is a drawback for a number of reasons. First, the server is now a single point of failure. In PGC if the server is compromised all that means is that messages might not be able to be sent. In TextSecure however, if the server is compromised, an attacker M could do a number of things to compromise messages. M could, for example, replace A's keys with his own, thereby letting B think that he is communicating with M. The only way for A and B to be sure that they are talking to each other would be to use TextSecure's QR generation and meet in person to verify. In PGC this scenario is not possible since the server does not store any keys.

2. While TextSecure, unlike PGC, seems to allow for sending secure messaging without having to meet in person, this is an illusion since any reliable verification needs to be done in person. In addition, with PGC after meeting in person once there is no need to meet up again to ensure scenario 1 has not happened since it is not possible in PGC.

3. Another attack that an attacker M can stage with TextSecure is to impersonate A by preemptively setting up an account with her phone number since TextSecure uses phone numbers to remotely identify users. B would request to communicate with A using her phone number but would in effect be communicating with M. In PGC users are protected from this since the users must exchange keys in person using their phones.

4. TextSecure's registration process is also vulnerable to a MITM attack where an attacker M can impersonate the server altogether and give B wrong information. In PGC this is not possible since the users exchange keys through the QR codes and have no need to communicate over a network to a server to exchange keys.

5. Other people [1] provide a concrete example of a similar attack on TextSecure. They describe the following Unknown Key-Share Attack (UKS): M knows that A will want to communicate with him. M then initiates a session using B's public key. He can justify the different public key by saying that he now has a new phone, or that he re-registered. If A now sends a message to M, M can just forward the message to B who will think the message was sent from A. In effect, B will think that A sent him a message when really A sent M a message. In PGC there is no danger of this UKS attack since there is no way for M to trick B into believing a message was sent from A since B will know that he never exchanged keys with A. In other words, TextSecure allows for unsolicited messages by relying on the public keys stored on the server. This is not a secure system, as is manifested by the described UKS attack.

[1] https://eprint.iacr.org/2014/904.pdf

5 Future Work

In this section be describes future changes and features that could be added to PGC.

5.1 Perfect Forward Secrecy

We consider the case of the private key being stolen as very unlikely. The users themselves are not able to access it outside the app, and no other adversary will be able to either. However, it is important to note that with the private key, the adversary could read any messages sent to the owner of the private key. Eventually, we plan on adding a new feature that will bring *Perfect Forward Secrecy* to PGC, which will use the public/private key pair to have clients agree on an ephemeral session key to encrypt messages with.

5.2 Anonimity

Currently, PGC offers users a high level of anonymity. Nowhere on the server is user information ever stored and an attacker that compromises the server cannot, from just looking at the server, see who is using PGC nor see who is messaging whom. However, an attacker could in principle see where messages are coming from and going to by looking at IP addresses or cell tower information. For example, the attacker could deduce that a user or a number of users from a particular geographic location communicates frequently with a user or users in another particular geographic location. While this is not necessarily something that most secure chat app users are worried about it is something that users do care about. Solving this problem would require different techniques, some of which will be similar to how Tor is implemented. This adds a different dimension to the current PGC implementation but could in the future be added to generate an even more anonynmous PGC.

5.3 Messaging UI Code Rewrite

In the current implementation the code that drives the chat UI (displaying message bubbles in the conversation window) was written in HTML/CSS/JavaScript. The JavaScript also uses JQuery to handle changes to the UI. This code can be presented in iOS using $UIWebView$. We have placed all the HTML/CSS/JavaScript code in a single file called `messages.html`. In order to remove the risk inherent in having the JavaScript make requests over the network we included all outside libraries in the code itself instead of making a network request. In fact, we also prevent any web requests from going in or out of the $UIWebView$ natively. There aren't currently any directly negative consequences of this implementation, however, due to potential Javascript updates in the future and the inherent flaws in the web security model, our next version of PGC would have the message text be displayed using native Objective-C/Swift and not rely on HTML/CSS/JavaScript at all.

6 Conclusion

We believe that future security will be based on mobile technology. Backend servers are very mysterious to users, can change at the drop of a hat, and depend on a range of even more third-party services. There is a lot of trust that users implicitly have by using other "secure" chat applications, and we created an alternative. By utilizing the security of the iOS ecosystem like Keychain and iCloud, we are able to securely store private keys that will be synced across all of users' devices, allowing users to have significantly less trust in and more privacy from a backend server.

Why do we trust an app more than a server? First apps are approved by store marketplaces, so they must meet the bare bones security requirements. Second, the app is something that changes far less and can easily be tested by security researchers by just analyzing outgoing and incoming

network requests. Additionally, we would publish the code as open source, and provide a signature of the compiled package such that researchers can trust that the code is actually what a user downloads.

Our contribution is twofold. First, we developed PGC which is an alternative messaging app that we believe provides better security and privacy than current chat apps on the market, while still maintaining good usability by creating an easy to use, fast in-app public key exchange. The second part is an analysis of *Telegram* and *TextSecure*, two of the current big players in secure chat applications.