Laszlo Makk
lm649@cam.ac.uk

Computer Laboratory, University of Cambridge
Summer 2016

Local Key Gossiping

1. Introduction

The project aims to explore and prototype gossiping public keys on a local wifi network. The Android (and later probably also iOS) application proposed by the project would aim to share public keys between users when their devices are connected to the same wifi network (just by running in the background), and detect key changes. It would do this while protecting the users' privacy to the fullest extent possible, and also in a highly automated way, possibly without any user interaction. The intention of the overall project is to create a key gossiping library that can audit the honesty of service providers (key servers), and can be easily integrated into existing secure messaging applications such as Signal or WhatsApp. The concept itself poses several problems to overcome.

This document aims to go through these problems, different parts of the protocol, and the implementation specifics of the prototype Android application, in varying detail.

2. Discovery

Devices need to find each other first, before they can have any kind of communication. We propose utilising the already established Bonjour protocol, commonly used to find printers on a local network without manual configurations. When the application is running in the background on a handset, the handset is advertising its Bonjour service (of a custom service type) with a random name, the local IP address of the device and a port number. Meanwhile, the app listens for incoming TCP connections on the advertised port. Alice can find Bob's registered service on the network by scanning for services of the predefined custom type and then establish a TCP communication on the IP and port she retrieves.

The app scans the network for registered services every few minutes. Specifically, at the beginning of a scan, a wake lock is acquired for 15 seconds, and just before releasing this, the next scan is scheduled to happen in two minutes. The wake lock is used to make operation more reliable by not allowing the phone to go to deep sleep while searching for other devices and carrying out the gossiping protocol with them. (To minimise battery usage, removing this wake lock should be considered, but properly establishing whether this causes significant problems would require further testing. Even if this turned out to be problematic, our tests suggest that for these purposes 15 seconds might be more than actually needed, and could potentially be reduced.)

The prototype uses the JmDNS library for the Bonjour protocol. We initially tried to use a competing implementation from the core Android framework itself (android.net.nsd.NsdManager), we encountered several issues, and ended up deciding against using it.

3. Sleep-related notes

While implementing the prototype, we encountered several "sleep" related difficulties. A handset goes to deep sleep soon after the screen turns off, which is essentially a lower power state for the CPU. This will result in almost all user-space programs stopping, in the sense that the JVMs would be paused and not execute instructions. To have some sort of guarantee that this won't happen to your application while in a critical state, you can acquire a wake lock, in which case the kernel code will know to keep the host JVM running. To keep the phone awake for your purposes, using the wake lock is the way to go.

What this application needs is to keep the device awake for a short time period, then release the wake lock to let the phone sleep and save battery, and then get rescheduled for execution some time later again. For this purpose, our prototype uses the android.app.AlarmManager class to set up alarms that the application can react to.

The Doze feature introduced in Android 6.0 restricts the amount of work apps can do when the phone is considered to be idle. Doze tries to reduce energy consumption by deferring background CPU and network activity. Doze defines two states/phases the phone can be in while in deep sleep, one in which all the restrictions are applied, and one in which applications can do work as usual ("maintenance windows"). These two states are alternating, but maintenance windows are less and less frequent the longer the phone is sleeping. This affects the prototype app in many ways.

Most notably, using AlarmManager, ordinary alarms (set with methods *set* and *setExact*) don't get called while the phone is in Doze, but Android provides new types of alarms which do (set with methods *setAndAllowWhileIdle* and *setExactAndAllowWhileIdle*). There are restrictions on these *"…AndAllowWhileIdle"* alarms however. While exact details are not well documented from official sources, we have found that an application is allowed a single *"…AndAllowWhileIdle"* alarm every 9 minutes. This effectively means the prototype scans the network every 9 minutes while the phone is in Doze.

This is not the full story though, as you are not guaranteed network access when the alarm is delivered. During tests, we have seen that scans stop finding devices (and hence no longer properly work) about 30 minutes after the beginning of the restrictive Doze phase, and they will only start working again when a maintenance window comes.

All this leads to conclude that currently the application can't really be relied on to work while the phone is in Doze. Worth noting however that entering Doze requires the phone to be disconnected from power, have a turned off display and be stationary, therefore it doesn't actually happen as often as one might expect. Although, Android 7.0 brings some changes in this regard, by introducing a third state for Doze with less restrictions than the original restrictive state, and entering this new state does not require the phone to be stationary.

Another "power-saving feature" of Android (also introduced in 6.0) is App Standby, which restricts the operations of apps that are not actively used by the user. The official Android documentation says[1] that an app is determined to be idle if the user didn't interact with it for some time and it doesn't have a foreground process or a visible notification (so essentially, the fact that the app is running is invisible to the user). Whether an app is determined to be idle/"inactive" at any given time can be checked in the Developer options (inside Settings, on the phone). We were originally expecting to run into this feature and planned to overcome it by using a persistent notification (showing a notification to the user that is always visible) but we ended up never finding the prototype listed as "inactive" in the mentioned listing, and hence decided to ignore the issue altogether.

[1] https://developer.android.com/training/monitoring-device-state/doze-standby.html#understand_app_standby

4. Setting up an encrypted connection

When a device discovers another, its primary goal is to transmit its public key. However, constantly revealing a user's identity on the network "in the clear" can be problematic as it facilitates user and activity monitoring, targeted man-in-the-middle attacks, etc.

Hence, we need a way for two instances of the application to decide if their users "know" each other, and only then send public keys. The decision process should leak as little information to others listening in as possible, and sending the public key itself should only happen on an encrypted channel. To accomplish this, we use the J-PAKE key agreement protocol, which can bootstrap a high-entropy shared key from a mutually known low entropy password (base secret) in three rounds, without essentially making any assumptions on the underlying communication channel. The Bouncy Castle crypto library provides an implementation of J-PAKE, which is what the prototype uses.

We are using phone numbers as the mutually known password. When Alice discovers Bob's Bonjour service and opens a TCP socket, she sends a ("salted phone number") message from which Bob can guess it is Alice if he knows Alice's phone number. The message sent by Alice contains a static salt, and k bits of hash(phone number, salt), along with k. Bob then computes the salted hash of every phone number (N) in his contact list, arriving at 0 <= n <= N "hopeful" phone numbers, among which may be Alice's number. Afterwards, Bob will start n instances ("handshakes") in parallel of the J-PAKE protocol with Alice, using the hopeful phone numbers as the base secret. If one of these has the correct (as in, the two participants are using the same) base secret, which would be Alice's phone number, that handshake succeeds; and they both derive the same high-entropy key to be used for symmetric encryption.

The number of revealed bits of the hash, k, has to be carefully chosen, as too few bits will result in a large number of J-PAKE handshakes, while too many bits might potentially result in revealing Alice's identity. The salt is static due to the consideration that an eavesdropper learns k bits of information regarding Alice's phone number for every message (of this type), but rather than being k new bits every time if a random salt was used, it is the same k bits. On the other hand, now the static salt itself is a static identifier for Alice, and everyone successfully gossiping with Alice can tie her to this identifier. The prototype uses k=9, as an average person having about 500 contacts seemed like a plausible guess.

When the J-PAKE handshake finishes, Bob sends Alice a message containing a port number (different from what he's advertising in the Bonjour service, which is what they had been using for the communication so far). Bob sends this message regardless of the outcome of the handshake (so even if it failed), so as not to leak the outcome to eavesdroppers. Alice then determines if the handshake was successful and decides accordingly to connect to Bob on this other port. They establish a new TCP stream, which is now going to be encrypted with AES in cipher feedback mode. The symmetric key for the encryption is the sha256 hash of the key material derived from J-PAKE. As this symmetric key is itself random, using a random initialisation vector did not seem necessary, so we ended up using a constant zero IV.

The prototype app uses the Java provided javax.crypto.CipherOutputStream and javax.crypto.CipherInputStream classes with a cipher obtained via Cipher.getInstance("AES/CFB8/NoPadding") for the symmetric encryption.

5. Sending public keys

After the encrypted connection is set up, Alice sends her public key to Bob. Specifically, Alice sends a message containing her public key, a timestamp, and the signed hash of (public key, timestamp, phone number). The hash used is sha256, and it is signed using Alice's private key.

The prototype app uses 4096 bit RSA keys, and signing/encrypting uses padding (OAEP). For these purposes too, we use the Bouncy Castle library.

Upon receipt, Bob first reconstructs the hash and tests the signature; he tests whether the hash is correctly signed, and whether it is signed by the corresponding private key of the public key just received. He also looks at the timestamp, and considers it valid if it is within 10 minutes of his clock. If one of these tests fails, Bob discards the message, and forcibly closes the connection.

Bob then checks if he has previous knowledge of any public key being associated with this phone number (that Alice claims to have). If this current public key is going to be the first public key associated with the number ("first use"), then Bob accepts the message. If this public key is found in Bob's internal DB with relation to this phone number ("update for already known"), the message is also accepted. If there is a different public key stored in the internal DB than the one he just received from Alice ("different than known"), the message is discarded, and the connection is forcibly closed.

If Bob accepted the message, he merges it into his internal database, which either adds it as a new entry or simply updates the relevant timestamps. He also stores the signed hash itself, making it possible to transmit a similar message to others later (with the "fixed" timestamp), and hence achieving transitive gossiping of keys (via these "history transfers"). Finally, Bob marks the communication channel as trusted.

6. Rate-limiting handshakes

Allowing an arbitrary number of handshakes to be carried out would have multiple consequences. Most importantly, that would allow online brute-forcing the J-PAKE base secret (the phone number). Also, as a J-PAKE handshake involves computationally relatively expensive operations, doing too many of them might result in a noticeable increase of battery usage.

In the prototype app, we are using two different rules for this rate-limiting. First, there is a global limit that over a rolling 30 minute time period allows 200 handshakes. Second, there is an individual limit that also over a rolling 30 minute time period restricts the number of handshakes per MAC address to 10.

The global limit affects both "incoming" and "outgoing" handshakes (as in, doesn't matter if given device is "Alice" or "Bob" in terms of the protocol), and also, it does not differentiate between successful and failed handshakes.

The individual limit on the other hand only limits one direction. Namely, Alice (who sends the salted phone number message) restricts the number of handshakes Bob can start with her. Also, it only aims to limit the number of failed handshakes, in the sense that the counter is reset if a J-PAKE handshake succeeds.

The individual limit mainly considers the incoming handshake direction due to it trying to limit malicious users. The global limit has two purposes, one being a second line of defence against malicious users who circumvent the individual limit by spoofing different MAC addresses, and the other is providing an upper bound for battery usage.

One potential improvement that comes to mind, that could have been implemented in the prototype, is an exponential punishment for the individual limit. That is, failing multiple handshakes would incur a longer and longer ban. It has to be noted though that this could backfire if sophisticated attackers spoof the MAC addresses of other honest users and devices end up banning those addresses for arbitrarily large time periods.

Also worth mentioning that a poor implementation of the individual limiting could lead to memory “leaks”. The prototype naturally tries to avoid this. Specifically, a map with MAC addresses as keys and linked lists of sorted timestamps as values is used to store the timestamps associated with J-PAKE handshakes. Every time there is a “request” to start a new handshake with a particular MAC address, the relevant linked list updated (older than length of rolling time interval timestamps get deleted, and potentially a new timestamp is added). The perhaps not so obvious addition to this is a periodic “garbage collection” that iterates through the whole map deleting too old timestamps (or complete lists if they become empty). Without this, the map could become quite large over long periods. The prototype runs a garbage collection upon starting a new J-PAKE handshake if it hasn’t been run in the last four hours.

7. History transfers

We not only want users to share their own public keys when they meet (when their devices are on the same wifi network), but rather, we would also like them to share and update their knowledge regarding their mutual contacts’ public keys.

Upon sending her public key to Bob, Alice creates a Bloom filter (using the Google Guava library) containing the phone numbers of all the contacts she gossips with (i.e. the contacts for whom she has key gossiping enabled). She then transmits this Bloom filter to Bob using the encrypted channel. Bob tests his phone book against the Bloom filter, and creates a list of possibly common contacts. Afterwards, Bob sends this list containing the phone numbers of the contacts to Alice. Alice then looks these phone numbers up in her internal public key database. For every phone number she finds, she creates a “public key flash” containing the phone number, a timestamp, and a signed hash of (public key, timestamp, phone number). The hash is signed by the private key corresponding to the public key of the owner of the phone number (if everything goes according to plan). Alice sends the list of public key flashes to Bob, who then refreshes the timestamps for the public keys he knows in his internal database.

Note that a public key flash is proof that the public key was still in use at the date of the timestamp (although not necessarily in use by the original owner if the key was lost).

Also note that a public key flash does not contain the public key itself (except for inside the hash). So, history transfers are only used for updating timestamps on keys, not for receiving and storing new keys we have not seen before. This is to make man in the middle-ing harder, by not allowing Alice to generate a new key pair, claim that it is Charlie’s, and send it to Bob; as this way Bob has to receive the public key from Charlie first. Although, if Alice knows that Charlie is a mutual contact for her and Bob, she can pretend to be Charlie by sending Bob a salted phone number message containing Charlie’s phone number.

Further note that if Alice sends a Bloom filter that claims to contain every phone number (full of ‘1’ bits) to Bob, then Bob will reveal all his contacts to Alice. This is one reason for the trust flag on the communication channel. The trust flag gets set to true after confirming Alice’s public key. If Bob receives the message containing the Bloom filter of contacts, and the trust flag is not set, the message is discarded. Still, if a malicious attacker guesses the phone number of one of Bob’s contacts for which Bob does not yet have an associated public key, the attacker can pretend to have that phone number, send a salted phone number message to Bob, successfully complete the J-PAKE, send a newly generated public key to Bob (which Bob will trust on first use) and then by sending a "full" Bloom filter, get Bob to reveal his complete list of contacts.

8. Phone number standardisation

The current protocol design and hence the prototype heavily relies on phone numbers being in some standardised format. Different devices are trying to reconstruct hashes where that hash content included a phone number of a contact, or phone numbers are being used as the basis of J-PAKE. Clearly, to make this work, we must represent the same logical phone number as the same sequence of characters everywhere.

For this purpose, the prototype uses Google's libphonenumber library. Specifically, the user first sets her own phone number inside the Settings view of the app. The entered string is sanitised in a naïve way such that only "+" and numeric characters are kept. The current implementation assumes that the user enters her number in an international format, that is, with the country code included. Then, when presented with another phone number to standardise, the libphonenumber library is used to do this. These numbers however don't need to be in an international format. If they aren't, the library infers the country code from the user's own local phone number and applies that to the provided number.

9. The internal address book

The app stores much of its persisted data in an SQLite database. Contacts (name, phone number) are kept in one of the tables. On first start-up, the user is requested to grant permission for accessing the Android system-level phone book/contact list. If the permission is granted all the contacts from there get copied over ("imported") to the app-level internal phone book (the SQLite db). Future periodic imports will also get scheduled on a 12-hour basis. If the permission is denied, the user can either grant the permission at a later time or start interacting with the internal address book directly (adding and deleting contacts there).

For every contact in the internal address book, an additional setting is stored. This is a three-way toggle that can be used to enable or disable key gossiping with that contact. Besides the enabled and disabled states, there is a third state named default which is what the contacts get set after being imported. This exists for the purposes of storing the (almost) additional bit of information whether the user explicitly set one of the extreme states for a given contact. The current behaviour of the default state is as if it was enabled.

If key gossiping is disabled for a contact, the corresponding phone number won't get be supplied to the protocol during any of the stages. That is, for example, that phone number will not be considered as a possible base secret for J-PAKE, will not be added to a Bloom filter, or included in a list of common/mutual phone numbers (as a reply to a Bloom filter).

Note that disabling key gossiping with someone however does not guarantee no key gossiping with that contact at all. Namely, if Alice disables key gossiping with Bob, but Bob enables it with Alice, then Alice will still send a salted phone number message to Bob upon discovering his service, and Bob will succeed with one of the J-PAKE handshakes, after which Alice sends her public key and does a history transfer. By the design of the protocol, Alice has no knowledge regarding Bob's identity; only Bob knows that he is talking to someone claiming to be Alice.

10. Limitations

The current prototype does not offer any solution to one of the main problems that key gossiping has to solve, namely key revocation. Users can reinstall the app, reinstall the phone OS, might even lose their phone, or just buy a new device (although this might be solved by migration/export-import but if the app is to be converted to a library, that might not be an option).

One scheme we have been considering is to have several partial keys that if combined can get you your original key pair. One such partial key could be stored in the Google cloud for example (Google Play Games Services – saved games, or Android Auto Backup) and the iOS counterpart, while the rest of them could be sent to a small set of trusted friends/contacts. A solution such as this would certainly work, but it would be problematic for a library, even more so for one that aims to work without any user interaction. Choosing the set of trusted friends might not necessarily require user interaction (though it most likely would) but getting back the keys from them in any sensible way most definitely would.