Zero Authorization Identity Proofs of Social Accounts for Mobile Apps

Alex Grinman
agrinman@mit.edu

1 Introduction

1.1 Problem

Many mobile apps today force users to create an account and authenticate before they can use the app. To make this process easier, apps allow users to authenticate with one of their existing social network accounts like Facebook or Twitter. To accomplish this, typically the user is redirected to the social network to authenticate. After successful authentication, the user is prompted to authorize the app to access some of the user's identity data. An OAuth authorization token is granted and returned to the app to prove the user's social identity.

Herein lies the problem. The app is granted continuous access to the user's data via the authorization token until it expires (currently 60 days for Facebook). In many cases the token is portable and grants access to unrelated information like a list of Facebook friends. Over time, privacy policies can change, and the user is always at the mercy of the social network that more of their data is not being released. There should be a way for a user to prove their social network identity without releasing any data that isn't already publicly available.

1.2 Cryptographic "Social Proofs"

Most social networking sites have a feature that allows their users to post messages that the entire world can see. This feature enables us to create indirect proofs of ownership for social networking accounts without asking the social network to issue any authorization tokens.

The idea is to use public-key cryptography to prove ownership. A "social proof" is a special message signed by the user's private key and posted publicly to the social networking site by the user. This special message contains the user's social networking account name and a time-stamp. This message proves that the user who knows the private key also has authentication access to the corresponding social networking account.

For example, suppose a user wants to create a proof for their Twitter account. The user would sign a special message and post the result as a "tweet" on their Twitter account so that everyone can see it. Now, when the user needs to prove to somebody that they own their Twitter account, all they have to do is provide a link to the tweet where the signed message is posted. The verifier would then independently read the tweet directly from Twitter and verify the signature with the user's public-key. The special message would include the user's Twitter username and possibly other information like a timestamp.

1.3 ZeroAuth Overview

We propose the "ZeroAuth" schema, which includes an API, an iOS app, and the social proof protocol.

When a user installs the ZeroAuth app and runs it for the first time, it will generate a public/private key pair, and securely store the encrypted private key. Next, the ZeroAuth app will assist the user in creating and publishing "proofs" for any supported social network using the social proof protocol.

Every mobile app that needs to verify the social identity of a user, will invoke the ZeroAuth API to communicate with the ZeroAuth app. Instead of the typical "Login with ...(Twitter/Facebook/etc)" button, there will be a "Prove Identity" button which will pass control to the ZeroAuth app.

The ZeroAuth app, through a callback interface, will send the user's public key along with a verification message signed by the private key back to the original app. The verification message will contain a random string. The ZeroAuth API will decrypt the verification message with the public key and once it verifies that they match, the app can now trust that the public key it received was in fact sent by the owner of the corresponding private key.

The ZeroAuth API will now obtain all links to available social identity "proofs" associated with user's public key. For each link, the ZeroAuth API will fetch the corresponding social identity "proof", and verify it using the public key. For every verified social identity, the ZeroAuth API will use the social network's website to fetch publicly accessible information about the user like their name, profile picture, and other attributes in an easy format for the app to use.

The result of this communication is that the app now trusts that the user owns their social identities without ever receiving authorization access to any of the social networks. Therefore, the user's personal data is not being compromised.

1.4 Deliverable: ZeroAuth Prototype

For my Undergraduate Advanced Project, I have developed and designed a prototype of the ZeroAuth Schema. For the purposes of the ZeroAuth prototype, the scope has been limited to Twitter accounts. The goal of the prototype is to demonstrate how any mobile app can ask users to prove their Twitter identity through the ZeroAuth schema without asking users for *any* authorization of their Twitter Account to *any* service (including ZeroAuth). All ZeroAuth app and API are built for the Apple iOS platform and several ZeroAuth design decisions, as explained later in the report, are chosen based on the security design of iOS.

The ZeroAuth app is not assumed to be trustworthy and therefore it is designed to verifiably prove everything to the ZeroAuth API. While the ZeroAuth API implementation provided is functional and follows the client-side schema, it is also not necessary to be trusted as implemented. The schema is openly defined and therefore any developer can create their

own ZeroAuth API implemenation.

The following are the components of the ZeroAuth Prototyped delivered in this project:

- Schema Definition. This is the ZeroAuth cryptographic protocol that defines the "Proving" and "Verifying" roles of the ZeroAuth app and $3^{rd}-$party app. This schema definition includes the protocol-level communication, explaining how the ZeroAuth app creates Cryptographic Social Proofs (Section 1.2) and proves them to the $3^{rd}-$party mobile apps that invoke the ZeroAuth API. This protocol explains how using the ZeroAuth API, a $3^{rd}-$party app can independently verify the correctness of the social identity proof independently of implementation details.

- ZeroAuth App. This is the direct interaction that users have with the ZeroAuth schema. The ZeroAuth app acts as an "identity" app that creates and maintains the user's public/private key pair, assists the user in creating Cryptographic Social Proofs, and implements the "Prover" role in the ZeroAuth schema. The identity app interacts with the $3^{rd}-$party mobile apps that seek to obtain a proof of the user's Twitter identity.

- ZeroAuth API. The ZeroAuth API implements the "Verifier" role in the ZeroAuth schema. The implementation of the ZeroAuth API takes the form of a client library that is responsible for both communicating with the ZeroAuth app directly through the operating system's inter-app communication protocol and verifying the identity proofs that ZeroAuth app returns. This library allows $3^{rd}-$party mobile apps to use the ZeroAuth schema so that user's can authenticate their Twitter accounts without any authorization.

- Demo $3^{rd}-$Party Mobile App. A ZeroAuth Demo app that utilizes our implementation of the ZeroAuth API client library in order to demonstrate how a user can prove their Twitter account identity to a $3^{rd}-$party app without authorization.

2 ZeroAuth Schema

This section explains in detail how the ZeroAuth app creates cryptographic Twitter proofs, and how using the ZeroAuth API, anyone can later verify these proofs. This section describes the communication between the ZeroAuth app and any other mobile app, seeking to verify a user's Twitter account identity using the ZeroAuth API.

2.1 Identity Creation

The first step is that the user must create their cryptographic identity in the ZeroAuth app. The ZeroAuth app, when opened for the first time, generates a public/private key pair and stores them securely in the device (explain further in section 3). For purposes of simplifying the security discussion for the ZeroAuth schema, here we assume that the private key is only accessible by the ZeroAuth app and no one else.

The following cryptographic functions are available to us in asymmetric key encryption: ENCRYPT, DECRYPT, SIGN, and VERIFY. We also use the MD5 hashing algorithm.

2.2 Creating a Twitter Proof

To create a cryptographic social proof for Twitter, the ZeroAuth app prompts the user to enter their Twitter *screenname* (i.e. `@AlexGrinman`) and uses the private key to generate the following message:

$$m = \text{MD5}(\text{SIGN}(screenname))$$

The ZeroAuth app then asks the user to independently post m (tweet the message m publicly). Once the user does, this the ZeroAuth app then uses the public Twitter API to find a tweet T which has the exact contents as m. The ZeroAuth app ensures that the author of T is the matching *screenname* and the contents match of the tweet match exactly m. Note that the MD5 above is needed since Twitter has a 140 character limit on tweets.

This tweet T is exactly the cryptographic social proof for the user's Twitter account. T proves that the user that knows this public/private keypair is also able to post to the corresponding Twitter account (i.e able to tweet from it). Therefor T is the linking proof between the public/private keypair and the Twitter account.

2.3 Proving the Twitter Proof to a $3^{rd}-$Party Mobile App

While the ZeroAuth app is convinced the user owns their Twitter account by T, a $3^{rd}-$party mobile app cannot because it doesn't know if the user in question actually has the associated public/private keypair. Therefore, in the protocol between the ZeroAuth app and any other $3^{rd}-$party mobile app, the ZeroAuth API must be able to verify that the ZeroAuth app knows the associated private key. We also want to ensure that this communication is not possible to be "replayed" by a malicious, adversarial app.

To accomplish this, ZeroAuth API generates a *nonce*, an ephemeral pseudo-randomly generated number, and sends it to the ZeroAuth app. The ZeroAuth app, signs the nonce by creating $s_1 = \text{SIGN}(nonce)$, and sends s_1 back to the ZeroAuth API along with the user's public key (denoted as `PUB`). The ZeroAuth API then uses `PUB` to verify that *nonce* was correctly signed in s_1. This part of the communication is an indirect proof that the ZeroAuth app knows the private key to the corresponding public key `PUB`, and it is secure against replay attacks.

The ZeroAuth API, next asks the ZeroAuth app for the social Twitter proof T. We define T_{ID} to be the unique ID of the tweet, generated by Twitter. Using the public Twitter API, anyone can directly look up this tweet (including contents and author) by just referencing the unique ID of the tweet T_{ID}. Hence, the ZeroAuth app sends over T_{ID}.

Finally, since the contents (m) of tweet T is a hash of the signature, the ZeroAuth app sends over $s_2 = \text{SIGN}(screenname)$, where s_2 is a signature of the corresponding user's Twitter screenname.

2.4 Verifying a Twitter Proof

In order to trust the information from the ZeroAuth app, the ZeroAuth API must do the following verification checks:

1. VERIFY$(s_1, nonce)$ with `PUB`
2. Look up T corresponding to T_{ID} **directly** from the Twitter API, read the following items from the Tweet:

$$screenname \leftarrow T.author$$

$$message \leftarrow T.contents$$

   where $T.author$ is the screen name of the user that posted T, and $T.contents$ is the exact message content of the Tweet.

3. VERIFY$(s_2, screenname)$ with `PUB`
4. MD5$(s_2) == message$

If all the above steps have succeeded, then the ZeroAuth API has successfully verified that the user has account access to the Twitter account *screenname*. Thus, the user, through the ZeroAuth app, has authenticated their Twitter account *screenname* to the $3^{rd}-$party mobile app using the ZeroAuth API.

2.5 Putting it all together

We now present Figure 1., a diagram illustrating the protocol followed by the ZeroAuth app and any $3^{rd}-$party mobile app utilizing the ZeroAuth API.

ZeroAuth app		$3^{rd}-$Party app/ZeroAuth API
		Choose a random *nonce*
	$\xleftarrow{nonce}$	
Create $s_1 =$ SIGN$(nonce)$		
	$\xrightarrow{s_1,\, pub}$	
		VERIFY$(s_1, nonce)$ with *pub*
Create $s_2 =$ SIGN$(screenname)$		
	$\xrightarrow{s_2,\, T_{ID}}$	
		Lookup T_{ID}: Get $T.author, T.contents$
		VERIFY$(s_2, T.author)$ with *pub*
		Check MD5$(s_2) == T.contents$

2.6 iOS Inter-app communiation

iOS does not have direct RPC-style communication between apps, but it does allow apps to invoke other apps with passable arguments. The communication between the ZeroAuth app and other mobile apps works as follows. When the user taps a "Login with ZeroAuth" button on the $3^{rd}-$party app, the ZeroAuth API invokes the opening the ZeroAuth app with the newly generated nonce and a callback for itself. Once the ZeroAuth app opens, it prepares $\{s_1, s_2, \texttt{PUB}, T_{ID}\}$ as shown above, and invokes the callback with these arguments,to go back the originating $3^{rd}-$party app, which then uses the ZeroAuth API to perform the verification step. More detail on the iOS inter-app communication protocol will be described in the next section.

3 ZeroAuth App

The ZeroAuth app is the user's direct interaction with the ZeroAuth schema. The ZeroAuth app acts as and "identity" app: maintaining the user's public/private key pair, assisting the user in creating cryptographic social proofs, and communicating with the ZeroAuth API to provide identity proofs.

3.1 First Time Use

When a user installs the ZeroAuth app on their iOS device and opens it, a public/private key pair is created and stored in the iOS Keychain (more details in section 3.3). **Figure**

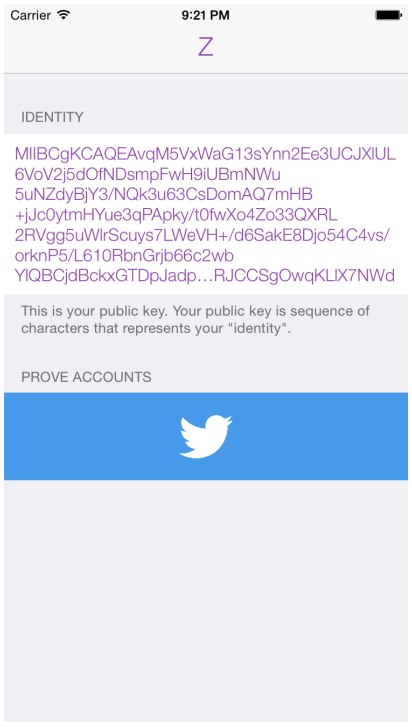

Figure 1: ZeroAuth app home screen, showing public key.

1 shows the home screen of the ZeroAuth app, with the following generated public key in `base64` encoding:

```
MIIBCgKCAQEAvqM5VxWaG13sYnn2Ee3UCJXlUL6VoV2j5dOfNDsmpFwH9iUBmNWu
5uNZdyBjY3/NQk3u63CsDomAQ7mHB+jJc0ytmHYue3qPApky/t0fwXo4Zo33QXRL
2RVgg5uWlrScuys7LWeVH+/d6SakE8Djo54C4vs/orknP5/L610RbnGrjb66c2wb
YlQBCjdBckxGTDpJadpMeAB5uFcBmXgmu1FGjCgJbpuQhxTRJCCSgOwqKLlX7NWd
Kz3FO2eqElxp0SZypxdY2/Km49ib5PJZiweReCPzV5HHYJiYVjslkMuTIae6snD1
bWmXJD6Yn95a+fzeHTQum5ukRj/6WM3uNwIDAQAB
```

3.2 Creating a Twitter Proof

The next step is to create cryptographic social proofs. For the purposes of the ZeroAuth prototype, only twitter is supported. The user taps on the twitter icon and enters their Twitter screen name as shown in **Figure 2**.

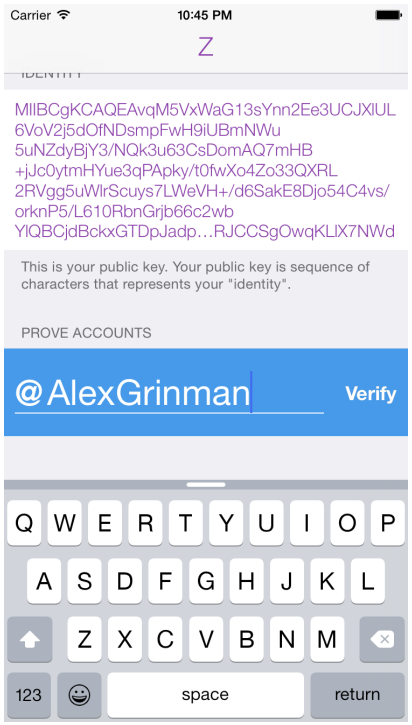

Figure 2: Begining to create a cryptographic Twitter proof

ZeroAuth uses this information to verify the user's Twitter account as follows:

1. Generate the hash of a cryptographic signature of the screen name (as explained in section 2.2).

2. Use the created hash and provided screen name to search for the publicly posted Tweet on Twitter. This verifies that the user own's the corresponding Twitter account.

After the user taps "verify", the ZeroAuth app asks iOS to prompt the user with a Twitter "share dialogue", pre-filled with the contents of the hash message generated from the signature of the entered screen name. Note that the share dialogue shown, as in **Figure**

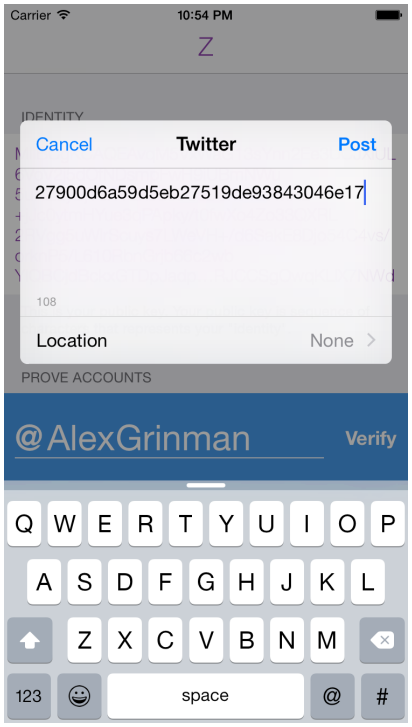

Figure 3: Posting the cryptographic Twitter proof

3, is a system contained window that even the ZeroAuth app has no control over (apart from providing pre-filled text and asking iOS to present it to the user). This share dialogue prevents ZeroAuth from ever requiring direct authorization. This part of the ZeroAuth app requires the user to have logged into Twitter inside iOS, which is acceptable since Twitter is built into iOS. However, even if the user does not want to login into Twitter via iOS, the ZeroAuth schema can still easily work, but instead the user will be asked post the generated hash to Twitter independently without direct assistance from the ZeroAuth app. We can imagine a simple option to copy the hash to the iOS clipboard, after which he user is responsible for posting it.

After the user taps "Post" on the share dialogue, the Tweet is posted publicly to Twitter. Next, the ZeroAuth app, using the Twitter search API, looks for the Tweet that has been authored by the screen name entered containing exactly the generated hash as the contents. Once everything has been verified, the ZeroAuth app shows that the user's Twitter identity has been successfully proved as demonstrated in **Figure 4**.

The user's Twitter profile picture, first name, and last name are pulled from Twitter, to confirm the user's correct identity. The Twitter icon now appears under "Verified Accounts", showing the verified user's screen name as well as a "fingerprint" button which links to the

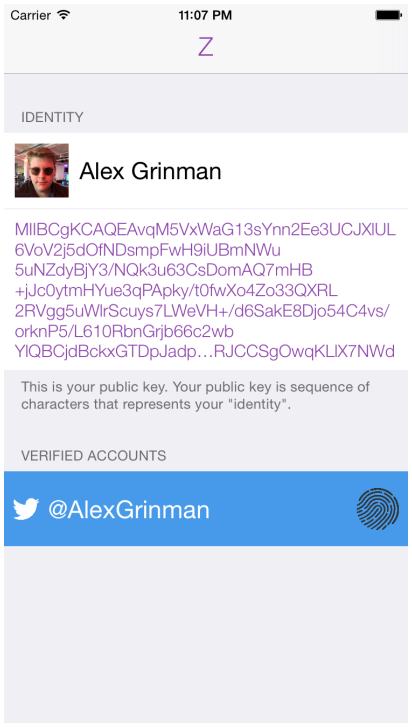

Figure 4: The Twitter proof has been verified.

posted Tweet, representing a "proof" of account ownership. For example, the proof Tweet depicted above is located at `https://twitter.com/AlexGrinman/status/584552583282429953`. The ZeroAuth app stores this proof url, which includes the T_{ID} (Tweet ID) appearing after `status/` in the url.

3.3 Securing the Private Key and Recoverability

The most important assumption that ZeroAuth makes is that the private key is stored securely. Luckily, the iOS Keychain, an encrypted database of for storing passwords and secrets, solves this problem for us.

The iOS keychain 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. The keychain cannot be read at all without unlocking the device, and we can employ a secondary TouchID/passcode entry screen that thwarts an attacker who takes the device when it's unlocked. By storing the private key in Keychain, this 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 sandboxed iCloud storage space) and resume normal use.

4 ZeroAuth API Implementation

As part of the ZeroAuth prototype, an implementation of the ZeroAuth API has been developed in the Swift programming language on the iOS platform. This library was utilized in order to create the ZeroAuth demo app presented in the next section. The code for the ZeroAuth API implementation follows the described protocol above, and some snippets are explained in the next section. The code is temporarily hosted in a private source control repository, but access can be granted upon request. Eventually, once complete, both the ZeroAuth app and the ZeroAuth API implementation will be completely open source.

While there can only be one actively used implementation of the ZeroAuth app, there can be any number of ZeroAuth API implementations as the protocol is also open. The ZeroAuth app cryptographically proves all of its knowledge to any ZeroAuth client, and therefore it is not necessary to trust a ZeroAuth API implementation since any developer can create their own. We provide an ZeroAuth API Client Library for the purposes of completeness and also to make it easier for developers to start using ZeroAuth in their apps.

5 Demo App: using the ZeroAuth API in a $3^{rd}-$party App

In this section we show a demo $3^{rd}-$party app that utilizes the ZeroAuth API. The purpose of this demo is to show how a developer would integrate the ZeroAuth API into their app, how a user will prove their social networking identities through the ZeroAuth schema, and a demonstration of the working ZeroAuth Prototype.

When the user first opens a $3^{rd}-$party app, such as this demo app, they should be presented with some sort of login button like the one in **Figure 5**. When the user taps this login button, iOS redirects the user to the ZeroAuth app. Note that once the developer includes the ZeroAuth API library (ZAClient) into their app, to invoke the login procedure the developer only need to call this one function:

```
ZAClient.sharedClient().requestIdentity("zademo://")
```

where "*zademo* : //" refers to the callback url scheme that the ZeroAuth app will use to return control back to the demo app. As part of this function call, the ZeroAuth API client library (ZAClient) generates a nonce and includes it as a parameter in the call to open the ZeroAuth app.

Once control is switched to the ZeroAuth app, the ZeroAuth app follows its part of the protocol outlined in section 2.5, creating and including the necessary parameters within the callback to the demo app.

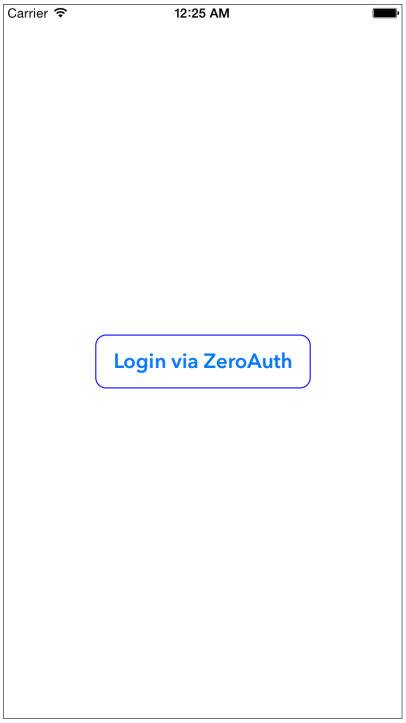

Figure 5: A ZeroAuth Login Button.

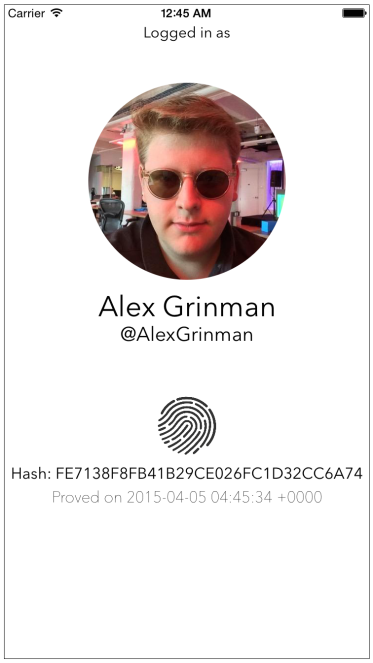

Figure 6: Control changes: *DemoApp* → *ZeroAuthApp* → *DemoApp*. Once the demo app is opened in the last step, it demonstrates the user's identity has been proven and verified.

Finally, the control switches back to the demo app. Every iOS app has a specific method where it handles invocations of its url scheme by the operating system. The developer only needs to include the following single line in that method:

```
ZAClient.sharedClient().handleIdentityResponseURL(url, source: sourceApplication)
```

The demo app, upon receiving the values as described in the protocol diagram of section 2.5, which are parameters of the url in the line above, verifies the components accordingly. Once the proof has been cryptographically verified and checked against Twitter directly, the demo app now provably knows the user's Twitter screen name and therefore their Twitter identity.

Thus, to demonstrate this, the demo app pulls some the user's publicly available (since it has no authorization access to private data) information like their profile picture and name. Additionally, the hash message is shown as well as the date of posted Tweet. The fingerprint button once again links to the direct proof, the posted Tweet on Twitter. **Figure 6** shows the final state of the demo app.

6 Related Work: Keybase

Keybase is an open source Public Key Infrastructure (PKI) that links a user's public key with their social identities and/or owned domains.

Setting up a Keybase account requires two steps. First, generate a public/private key pair on a local computer and post the public key to the Keybase server. Next, use the Keybase client to sign a message containing the public key and post the message on supported social networks (currently Twitter, GitHub, Reddit, and Coinbase, etc...) where it is publicly readable but only writable by the user. Therefore, this signed message is a "proof" that the user owns this social identity.

Keybase maintains a directory of public keys, where public keys can be searched by social network account names. Each directory entry contains the public key of the user and links to the user's social identity proofs. These links can be used for identity verification by any apps independently from Keybase. For example, for Twitter the link is to a tweet like `https://twitter.com/alexgrinman/status/507198975334973441` and for GitHub it's a link to a public "gist" like `https://gist.github.com/agrinman/edd1af773f3561f8ee47`.

6.1 Similarities and Differences with ZeroAuth

Keybase's goal is to be a public, searchable PKI that lets users verifiably find the public keys of their social networking contacts. ZeroAuth, on the other hand, uses cryptographic identity proofs to allow user's to authenticate their social networking accounts without providing authorization to any involved party. ZeroAuth aims to be a "zero authorization" identity app and protocol, operating independently of identity providers.

ZeroAuth also focuses on usability, taking care of all the cryptographic related work for the user as well as providing an easy, understandable interface. ZeroAuth users do not have

to have any technical knowledge, and they can rely on ZeroAuth to assist them in creating proofs and storing the private key. Keybase, in its current state, requires its users to be very technical: understanding PGP, generating public/private keys, storing public/private keys, and creating social proofs.

7 Future Work

In this section we describe the next steps needed to advance our ZeroAuth prototype into a usable, deployable, zero authorization schema for proving account ownership.

7.1 Android and Web

For ZeroAuth to be successful, it must be accessible on popular platforms. A big next step will be to add support for both Android and Web. One of the key aspects of ZeroAuth is that the private key never leaves the device. Therefore, a web solution cannot implement the role of the ZeroAuth app. However, we can still provide a web based ZeroAuth API client library that interacts with the ZeroAuth app. Since the ZeroAuth app never reveals any private information to clients, and because it adheres the simple iOS url scheme communication protocol, we can easily build a ZeroAuth API implementation for web.

In Android, on the other hand, we are dealing with a device, but securely storing the private key is not as easy as it is in iOS. Android doesn't offer such a strong secure key database like the iOS Keychain. Instead, the Android version of the ZeroAuth app could utilize the chrome password storage service. More research will need to be done into the security guarantees of this service.

7.2 Expanding to other Social Networking Services

While the presented ZeroAuth prototype only supports Twitter, in the future, other social networks like Facebook, GitHub, Reddit, and more will be supported.

7.3 Beyond Social

We can expand this schema to go beyond just social networks. Any identity provider can create server space where only authenticated users can post content that is publicly readable, and thus the ZeroAuth app could provide a proof of identity without trusting any $3^{rd}-$party authority, entering a password, or issuing authorization access tokens.

ZeroAuth is not only a "Zero Authorization" schema, but it is also a "Single sign-on" solution for mobile apps because the user enters credentials only once when they authenticate with identity providers in order to post proofs.

References

[1] Apple. *iOS Security Guide.* `https://www.apple.com/business/docs/iOS_Security_Guide.pdf`