6.858 Final Project: Distributed Authentication on the Ethereum Blockchain

Michael Shumikhin, Sarah Wooders, and Ryan Senanayake

Abstract— **We introduce KeyChain, a trust-less authentication system which stores username to public-key mappings on the Ethereum blockchain. Keychain includes ann account recovery feature, where users set up a "web of trust", where users can authenticate other users via the mapping and recover their accounts if k of n members of their web of trust approve. We implement a prototype of Keychain using an Android app to control a user's account and store keys, a Node.js app as an example application to authenticate with KeyChain, and the Ethereum blockchain to record name/key mappings and the user's web of trust.**

I. Introduction

The current state of digital authentication requires significant trust in 3rd parties. Users need to trust websites to store their passwords safely. Websites need to trust users to choose secure unique passwords, which the majority of users will not do. And even for the minority of users who do, their account can still be compromised by trusting recovery emails, computers with malware, companies, etc. KeyChain aims to improve upon current authentication systems with the following features:

- *Trust-less authentication* Users do not need to trust KeyChain or any websites they authenticate with.
- *No passwords*: Confirmation on a trusted device provides access.
- *One account*: One KeyChain account allows for access to all website accounts
- *Safer recovery*: Users can recover forgotten private keys or revoke access for compromised keys without compromising their security by relying on recovery emails or security questions.

Recoverability is a crucial aspect of a useable authentication system. If it is too difficult for a user to recover from losing or forgetting a secret, there will be no adoption of a system even if it is more secure. This is an issue with public-private key encryption, which offers no ability to recover from a lost private key file.

Below is outlined a comparison of our system with a few different methods of authentication:

A. *Website stores hash of password*

A User has to trust the website to store their password correctly, which often is not the case and results in key compromise. Websites also trust that users will choose unique, high-entropy passwords, which for the large majority of users is not the case. Additionally, many sites will provide a forgot your password option, which relies on security questions or a recovery email, both of which may be even less secure than the password.

B. *Sign in with Google, Facebook, etc*

This requires trusting the company behind the authentication platform, which may not be a big deal for the majority of users, but it does mean that there will likely never be one solution that everyone uses. For example, Apple would probably not like the idea of trusting Google for their authentication system. Additionally, these systems still rely on the user picking a unique, high-entropy password. Best practices also dictate that a user make these hard to remember passwords different. While it is easier for a user to make one good password, it is often easier for a user to make a memorable username which may be protected with public key cryptography.

C. *Providing public key to website*

While this is a much more secure system, if you lose your private key it is not possible to recover. Furthermore, this is not a prevalent means of authenticating to a website as it requires a separate challenge and response client on the device to respond to authentication requests from the website. Allowing multiple private keys to authenticate a public user/record provides significantly more flexibility.

D. Secret Sharing

Secret sharing allows for a private key to be distributed in pieces to form a web-of-trust like recovery mechanism. This could be used in conjunction with providing a public key to the website to allow for recovery from losing a private key. A significant disadvantage of this system, however, is that if a private key becomes compromised, there is no way to change the private key. Therefore, an attacker would still have full access to the account.

E. NameCoin

Namecoin allows users to claim identifiers, associate public keys with those identifiers. Namecoin allows users to purchase ownership of any number of unclaimed names for a set fixed fee per name. KeyChain addresses several usability and adoption shortcomings in Namecoin. The primary usability limitation is that if users ever lose the private key used to register an identifier , they will permanently lose control over that identifier. With KeyChain's web of trust, recovery is possible and built into the system. Similarly, if a key is compromised, an identifier can be permanently and irrevocably hijacked. In keychain this is avoided by allowing any associated key (prior to web of trust recovery) to revoke the set of associated keys for an identifier. Thus, KeyChain requires significantly less key management effort and are not excessively burdened by responsibility. As any Ethereum address can participate in the KeyChain authentication system and since Ethereum is significantly more utilized than Namecoin, we believe Keychain is more amenable to adoption.

II. Threat Model

Our primary goal is to provide robust username-based authentication without exposing private keys or relying on 3rd parties. We assume that users store their keys on a trusted device and that private keys are only exposed for the signing of challenge-response and at least n-k of designated recovery users are trustworthy. We assume the attacker can compromise the website, the network, and fewer than k recovery users for a given user.

We develop Keychain system based on a mobile app for the user-facing side, similar to the DUO mobile app.

III. Approach

A. Authentication

We build a prototype for Keychain with three main components: A mobile app for account management and key storage, a smart contract for interfacing with data stored on the blockchain, and a dummy webapp which contains a KeyChain authentication form. A new user will create a KeyChain account by downloading the KeyChain android app and making an account, which will generate keys to be stored on their mobile device. Keychain is based off Elliptic Curve cryptography, making the assumption that users will have an Elliptic Curve private key securely stored on their physical devices (e.g. phone or computer). Users can then authenticate with a web application by clicking “Login with KeyChain” on the website, which will redirect them to a QR code page. The user scans the QR code from their mobile device, which provides them with a nonce and callback URL to the webapp. The callback URL is called from the mobile device to send data containing the user's public key, the signed nonce, and keychain ID to the webapp. The webapp verifies the keychain ID to public key mapping with the Ethereum blockchain as well as that the nonce was signed by the public key. Once this is verified, the web server provides an access token to the client's browser. An overview of the system can be seen in **Figure 1**.

B. Web of Trust

Whoever is trusted by k out of n members (where k is chosen by the user) of the web of trust is defined to be the correct user of an account. Our system does not account for a malicious web of trust. When the user creates an account, they designate an initial web of

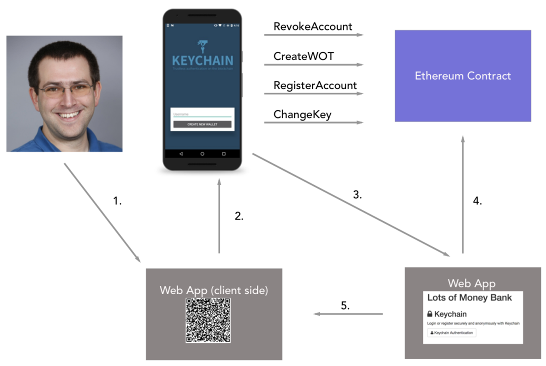


Fig. 1. The user downloads the Keychain mobile app to register an account, create a web of trust, change their keys, or revoke their account. The authentication process is as follows: 1.) A user requests to log into a webapp, and is shown a QR code. 2.) The user scans the QR code with their mobile app. 3.) The mobile app sends the signed nonce and Keychain ID to the web server. 4.) The web server verifies the public key to Keychain ID. 5.) If verified, access token provided to client.

trust as well as a starting public key. Once this is formed, the user no longer has the power to change their web of trust. The web of trust should be able to elect a new web of trust (though this is not in the current smart contract implementation).

Once this trust anchor is established, the web of trust can at any time vote to wipe the user's account and start fresh with a new public key. Any user with a public key that was added since the last web of trust vote also has the power to force a web of trust vote by freezing the account (see section 5.5 to see how this helps mitigate the effects of a compromised key). Note that even a public key that was removed and therefore does not have login access can still force a vote. A public key with current login privileges is allowed to add or remove other public keys to allow for the user to add other devices or delete a public key when they lose a device.

IV. Implementation

We provide the details for the three main components of Keychain. We implement the mobile app as an Android app, our smart contract using Solidity for the Ethereum blockchain, and a dummy webapp using Node.js.

A. Android App

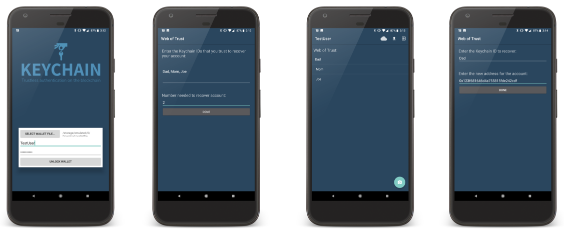
The Android is the primary method for account management. Users create their Keychain account using the Android app. Users can import an existing Ethereum wallet key file into the app (this file is always transmitted encrypted by a user-known password. A future implementation would also allow the user to generate a new key pair within the app so it does not leave the device. Importing an Ethereum wallet provides the convenience that this wallet is already funded and therefore can pay for the transaction fees on the network. We experimented with methods that did not require the user to pay transaction fees, however, we think that this is important to prevent name-squatting. We think that this problem will be less severe in Keychain than in Namecoin as the meaning of identifiers don't have intrinsic value unlike domain names.

Once the user creates an account, they are prompted to add other users to their "web of trust". Once they are logged in, they can vote for a reset of another account's public key via the web of trust. They can also authorize new public keys for use on different devices. All of these interactions are transmitted to an Ethereum node run by Infura, which sends transactions to interact with the smart contract. The most heavily component of the Android app is the QR scanner that will send the response to the callback URL. We had a difficult time replicating the Ethereum elliptic-curve cryptography operations on Android and had an even harder time making sure that our node.js library was accepting the same encoding of the signature as Android was creating. However, we were able to do this by using cryptography operations provided by a third-party library and sending the components of a signature separately to the web server.

B. Ethereum Smart Contracts

We implement a smart contract in Solidity to store a trusted mapping from user ids to a set of authorized public keys and manage accounts and keys. We include the following methods:

```
Do_add_key(bytes32 user, address
    new_key);
Do_add_key(bytes32 user, address
    new_key);
```



/storage/emulated/0/

TestUser

Enter the Keychain IDs that you trust to recover your account:

Dad, Mom, Joe

Number needed to recover account:

2

Web of Trust:

Dad

Mom

Joe

Enter the Keychain ID to recover:

Dad

Enter the new address for the account:

0x123f681646d4a755815fde242cdf

Fig. 2. Users can register and account, add members of their web of trust, and revoke their account using the Keychain Android app.

```
Do_recover_address(bytes32 user,
    bytes32 recovery_user, address
    new_address);
Do_recover(bytes32 username,
    address new_address);
Do_revoke(bytes32 username);
```

New users are added to KeyChain using the `Create_username` method. The `Do_add_key` function is used to associate more keys with a username. The trustless storage/computation offered on the ethereum blockchain is able to make updates to this state. A user-id can map to many public-private key pairs and likewise a public-private key pair can map to many user-ids. This provides a user with a significant amount of flexibility to silo user-ids by device, public-key pair, and 3rd party service.

Recovery users are stored as an immutable array of user-ids whose constituent public keys can designate a new public key for a compromised user-id. If any of a user's private keys are compromised then members of the user's recovery group can designate a new recovery public key for the user by calling the `Do_recover_address` function with a new recovery address. Once a pre-specified k of n majority of recovery users have designated a recovery public key, the contract re-assigns the user-id to public key mappings to the new key after anyone runs the `Do_recover` function. The contract is able to revoke the set of key pairs associated with any key currently associated to a user-id using the `Do_revoke` function.

C. Node.js Front End

We create a dummy website that uses Keychain authentication. We use the web3 library for Javascript to interface with an ethereum node run on Infura. This node allows us to interface with the Solidity contract. Users authenticate with the webapp by clicking "Log in with Keychain", which directs them to a QR code image contains a nonce and callback URL for sending data back to the web server. On receiving this QR image, the browser receives a signed token by the website that authorizes it for use of this nonce. The web server ensures that it gives unique nonces to each request. The mobile app scans this image and then signs $\{nonce, KeychainID\}$. The mobile app then sends this signed message as well as the public key used to sign to the callback URL. Once the web server receives this request, it verifies that the nonce was signed by the public key. Then, it verifies the Keychain ID and public key by calling `Query_user_keys(bytes32 username)` from the Solidity contract. Once verified, the client can send its signed token and exchange it for a typical authorization token for that user's account.

V. Evaluation

KeyChain is built to be secure without relying on trusting any centralized entity. We outline some of the threats that we are most worried about in this section.

A. Compromised Website

A typical threat that affects passwords consists of an attacker who has gained access to a website that the user has an account with. Depending on how the password is stored, the attacker may be able to recover the user's password and have access to all accounts that share the same password. This relies on trusting the website to store your password correctly. Ideally users would use different passwords on each website, but realistically this does not happen.

In KeyChain, the attacker can only recover the user's KeyChain id. With this information, the attacker can only discover the user's public key, which does not compromise any other accounts owned by the user. In addition, we assure some level of confidentiality as an attacker is only able to associate this account with other sites that make the uncommon choice to make the user's KeyChain ID public.

B. Phishing

Among the most common and costly attacks today, phishing is a persistent problem for most services. Certificates, spam filters, and anti-phishing images/codes have been able to protect many users, but inherently visiting a compromised or similar-looking website allows for a MITM attack to the real website that the user intends to authenticate with. As the website has a trusted mapping between users and public keys, this allows for a secure channel to be created with the user that cannot be MITM'd. This will not work with the Android app, but would work with a browser extension (not included in current implementation) that can decrypt the received token for the browser.

C. Altering the KeyChain ID to public key mapping

If an attacker was able to alter this mapping, they could substitute their own public key and proceed to take control of a user's account. In a centralized service, we would worry a lot about mechanisms for user's to check the integrity of this central database, but we can rely on the immutability property of blockchains to protect this mapping. As long as our smart contracts are implemented correctly, changing this mapping would require a cryptographic attack on a user's public keys.

D. Brute-force attacks

We use the same cryptography scheme as Ethereum and so if a cryptographic attack was possible on our public keys, then every wallet on Ethereum could be compromised by the same attack. We maintain the same brute-force resistance as any other Ethereum key-pair based on the secp256k1 elliptic curve and KECCAK-256 hash function.

E. Physical access to a user's device

While we are most worried about low-cost remote attacks, KeyChain also guarantees some security from an attacker with physical access to a user's device. On Android, the private key is stored as an Ethereum wallet file and encrypted with a password on disk. This password is stored securely by the application and used to decrypt the wallet once the user has provided their fingerprint or device pin code.

We are still worried about an attacker using the user's device to add a new public key to their KeyChain ID then could then remove the victim's public key so that that they become locked out of their account. This would require an attacker having possession of a user's device and knowing their pin, pattern, or fingerprint to authorize the creation of a signature. With this information an attacker would also have access to the user's email accounts which they could use to change the password on all websites that use this as a recovery email.

We also provide the ability to recover from this attack. Our smart contracts allow for any public key that was ever added to an account (even if it was removed by an attacker) to freeze the account. This revokes all current public keys associated with the user's account. Once this has occurred, the user must request the web of trust to create a new public key. Therefore, even if the attacker has a compromised private key, they only have access to the user's account until the victim notices the attack and locks their account. This is much better than a scheme where the user directly provides their public key to a service as in this scheme the attacker becomes indistinguishable from the victim and therefore has unbounded access to their account.

F. Attacker on the network

As the response is coming from a different device than is being logged in, extra care must be taken to ensure that an attacker does not authenticate herself with an existing challenge response. As an example, a possible attack is:

1) The user attempts to login and creates a challenge
2) The user's browser starts polling the server for the result of the challenge
3) An attacker starts polling for the same challenge response
4) The user completes the challenge
5) Both the attacker and user are logged in

To prevent this attack, the NodeJS Integration Library ensures that each random nonce is only provided to one user and is given a signed token to prove that it should be signed in when the challenge completes. As long as this token is transmitted over a secure channel, this ensures that the system is no longer vulnerable to this attack.

G. Compromised Application Code

An attacker could potentially introduce vulnerabilities in our application code and distribute this to users if they obtain our publishing keys for the Android and Chrome app stores as well as our publishing account information. Publishing keys will be kept offline to help deter this threat. We will also be open sourcing all of our code so that anyone can choose to build their own versions of the applications. Finally, our applications are only reference implementations and anyone could choose to make their own implementation, which can still interact with our smart contracts.

VI. Conclusion

Our design addresses flexible user-id recovery, key revocation, phishing resistance, and is MITM resistant. We believe that the privacy of user-ids and associated recovery users can be improved upon in future iterations using hashing for privacy or a similar method. A weakness of our design is that users/services cannot easily transition to use KeyChain and that transactions require Ethereum to fund which is not easily available to end-users. Overall, we hope our design provides advantages in the area of user authentication compared to current alternatives.