Portis: key management and smart contract interaction using end-to-end encryption

Itay Radotzki Tom Teman

June 11, 2019

Abstract

Using end-to-end encryption would allow users to maintain absolute and sole control over their private keys and to interact with smart contracts, without the risk of losing access to their wallet if they misplace their device containing the only copy of their private key, while enjoying a "cloud-like" experience, as users will be able to access the same wallet from different devices, using a familiar email and password login mechanism from their existing browsers, without installing any third-party applications. Other solutions in the ecosystem introduce friction into the onboarding process by placing the burden on the end user, requiring them to install their third-party solution before they can begin using a Decentralized Application (DApp). Portis aims to eliminate that friction by offering developers an SDK. Once integrated, users are able to manage their private key inside their existing browser, letting them sign transactions and manage their crypto assets. At the same time, they remain the sole owners of their private keys, as our end-to-end encryption architecture makes sure nobody can access the user's unencrypted private key - besides the user itself.

1 Introduction

By offering complex functionality via smart contracts, DApps can leverage the advantages of blockchain technology whilst also being able to provide an easy-to-use product or service. An essential component of DApps is a provider which allows users to securely store their private key and use it to sign transactions that are relayed by the provider to the blockchain. Private key management is a difficult challenge for tech-savvy individuals, let alone for average users. One approach is using a service which acts as a custodian of the users' private keys and signs transactions on their behalf, using a standard OAuth architecture to determine access to said private keys. The problem with this custodial approach is the "If You Don't Own Your Keys, You Don't Own Your Crypto" adage. If the service is hacked, destroyed or malicious, the user's funds could be stolen or lost forever. The ethos of the blockchain ecosystem is that users should be the sole owners of their private keys. So although this approach seemingly offers an easy solution to the challenge of key management, offering simple key recovery mechanisms, it is counterproductive to the goal of decentralized apps, preserving the same issues stemming from centralized services.

Another approach is storing the private key inside the user's device. While this key management solution does make sure users are the only ones who have access to their private key and is non-custodial, it introduces the risk that users might misplace their device, thus losing access to their wallet forever. In addition, the concept that your digital identity and value is tied to a specific device, and that losing that device would mean forfeiting both, is a foreign concept for the average user, which is used to cloud services, meaning they can always access their account as long as they have their login credentials. In this day and age, losing your smartphone does not equate losing access to your bank account, social networks, etc. Furthermore, if a user wishes to access the same wallet on multiple devices, they need to manually import their private key, which can be an intimidating process for the average person. Once more, users are accustomed to being able to access the same account on multiple devices in an easy and familiar manner.

If we want to reach mainstream adoption, we need to offer users a solution which feels just as simple as using web2 applications, without compromising the underlying principles of decentralization or security.

We propose a key management solution that employs end-to-end encryption. Portis lets users enjoy the best of both worlds, as they remain the sole custodians of their private keys, yet don't need to worry about losing them forever if they misplace their

device. In addition, since their encrypted private key is stored on the cloud, they can access the same wallet on multiple devices using a familiar email and password login flow.

Finally, while other solutions demand the user installs a third-party plugin, application or browser to use any decentralized application, our approach is to instead ask the developers to integrate our SDK with a few lines of code, removing that burden for their users, without compromising security or making DApps or any other service the custodians of private keys.

2 Components

Portis is more than a cryptocurrency wallet, it also enables users to interact with smart contracts. Once the Portis SDK has been integrated into a DApp, it lets users sign transactions and messages with their private key. It adheres to the standard protocols of doing that for each blockchain (for example, when used in an Ethereum DApp, Portis generates a standard web3 provider instance). The private key never leaves a user's device in its unencrypted form.

2.1 Client

The encryption and decryption of the user's private keys happen in only two locations. The first location is the Wallet, which is a web application that runs under the portis.io domain and allows users to view, send, receive and purchase crypto assets. In addition, users can also export their private keys, import pre-existing private keys, set up 2FA, and more. The second location is the Widget, another web application that runs under the portis.io domain. The Widget is where transactions and messages are signed when users are interacting with a DApp. All of the transactions and messages that need to be signed, as well as all other calls to the blockchain, are received from the SDK and are then relayed to the blockchain from the Widget, through the node that was defined when the Portis instance was initialized in the DApp. The resulting transaction hashes are then relayed back to the SDK, which in turn exposes it to the DApp.
Since the underlying encryption is the same for both the Wallet and the Widget, for the sake of brevity we will treat them as one in this technical paper, and refer to them as the Client. The Client is where the end-to-end encryption takes place and is the only location where the user's private key exists in its decrypted form.

2.2 SDK

The Portis SDK is an open-source library that acts as a proxy between the DApp and the Widget. It is a gateway that lets DApps communicate with the blockchain using the relevant standard protocol as well as generate transactions and messages to be signed inside the Widget.

2.3 Communication

The code of the Portis web SDK is hosted under the DApp domain. It generates an iframe pointing to the portis.io domain, in which the Client code is loaded. The SDK and Client communicate with each other using the browser's native postMessage mechanism.

3 Encryption

Portis lets users create an encryption key on their Client, so once they generate blockchain wallets on their devices, they will be able to encrypt them using said encryption key, and store the encrypted wallets on the Portis servers.
All cryptographic keys are generated and managed by the user on their devices, and all encryption is done locally in the Client. Portis servers are never in the position of learning your cryptographic keys. When the already encrypted data travels between the user's device and our servers, it is encrypted and authenticated by TLS. All of the user's sensitive data is encrypted when they create their account using 64 random bytes generated on the Client, protected using a password that they select. Nobody on earth knows this password besides them as it never leaves the Client. Using a KDF algorithm, a Backup Recovery Phrase is derived from the password, to offer users a means of resetting their password in case they forget it. We do not have the ability to recover users' data if they forget their Password and lose their Backup Recovery Phrase (because of end-to-end security).
We strive to bring the best security architectures to people who are not themselves security experts. Not everyone wishes to become a security expert and most will not read this document, but everyone is entitled to security whether or not they seek to understand how it works. Concealing the necessary complexity of the design from users when they just want to use a DApp or manage their cryptocurrency funds is all well and good, but we should never conceal the security design from security experts, system and security administrators, or curious users. If the security of a

system depends on some aspects of the design kept secret, then those aspects would actually be weaknesses. Therefore, we strive to be open about how our system works as much as possible.

We use the browser's native Crypto interface of the Web Crypto API for the following cryptographic functions: *getRandomValues*, *importKey*, *deriveBits*, *digest*, *sign*, *encrypt*, *decrypt*, *exportKey*, *generateKey*.

3.1 Create Account

Step 1: The User generates a secret. User inputs an *email* and a *password* into the client.

Step 2: The Client generates an encryption key. The client generates an *encryptionKey* of 64 random bytes using the browser's native function *Crypto.getRandomValues()*.

Step 3: The Client encrypts the encryption key. The Client selects a Key Derivation Function (KDF). The default choice is PBKDF2. PBKDF2 is a password-based key derivation function that uses a password, a variable-length salt, and an iteration count and applies a pseudorandom function to these to produce a key. Our implementation uses SHA-256 as the pseudorandom function.

The Client selects the number off KDF iterations to run. The default choice is 100,000. The client runs the KDF with the *password* as the secret, the *email* as the salt and the relevant number of iterations. The KDF function returns a key, which we will call *passwordDerivedKey*.

The Client runs the AES-CBC encryption algorithm, with the *encryptionKey* as the plaintext, the *passwordDerivedKey* as the secret, and 16 random bytes as the IV (using the browser's native function *Crypto.getRandomValues()*), returning the *encryptedEncryptionKey* cipher, composed of the *encryptedEncryptionKey*, the IV and the encryption algorithm mode.

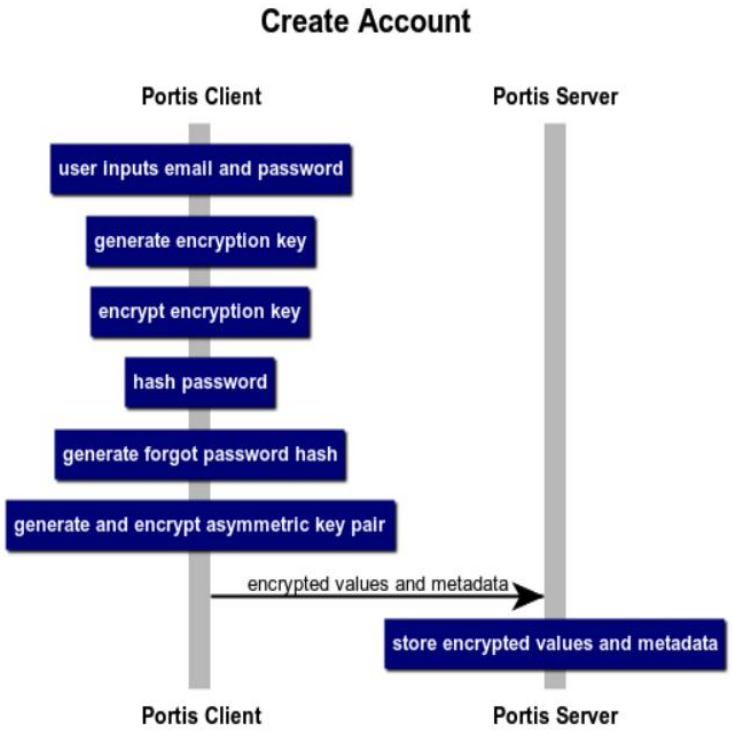
Step 4: The Client hashes the password. The Client runs PBKDF2 with SHA-256 as the pseudorandom function, using the *passwordDerivedKey* as the secret and the *password* as the salt, running a single iteration. This returns the *passwordHash*. The *passwordHash* serves as the Client's authentication key against the Server.

Step 5: The Client generates a forgot password hash. The client hashes the *passwordDerivedKey* using SHA-512, returning a *passwordDerivedKeyHash* (this will be used for the "forgot password" flow).

Step 6: The Client generates and encrypts an asymmetric key pair for secure client-to-client communication. The Client runs RSA-OAEP 2048 with a SHA-1 hashing algorithm to generate a pair of both an *asymmetricPublicKey*

and *asymmetricPrivateKey*. The Client runs the AES-CBC encryption algorithm, with the *asymmetricPrivateKey* as the plaintext and the *encryptionKey* as the secret, returning the *encryptedAsymmetricPrivateKey* cipher, composed of the *encryptedAsymmetricPrivateKey*, the IV and the encryption algorithm mode. For now, these *asymmetricPublicKey* and *encryptedAsymmetricPrivateKey* have no function but will be used later on as asymmetric keys for secure client-to-client communication.

Step 7: The Client stores the encrypted values on the Server. The Client sends the following data to the Server, which stores these values: selected KDF, selected number of KDF iterations, *email*, *passwordHash, passwordDerivedKeyHash, encryptedEncryptionKey* cipher, *asymmetricPublicKey,* and *encryptedAsymmetricPrivateKey* cipher.



3.2 Login

Step 1: The Client fetches encryption metadata from the Server. The User inputs their *email* and *password* into the Client. The Client calls the relevant Server API, passing along the *email*, and receives back the following values: *KDF* (the type of Key Derivation Function used by the Client during the create account flow), *KDF_ITERATIONS* (number of KDF iterations used in the KDF process during the create account flow) and *USER_VERSION* (for internal use).

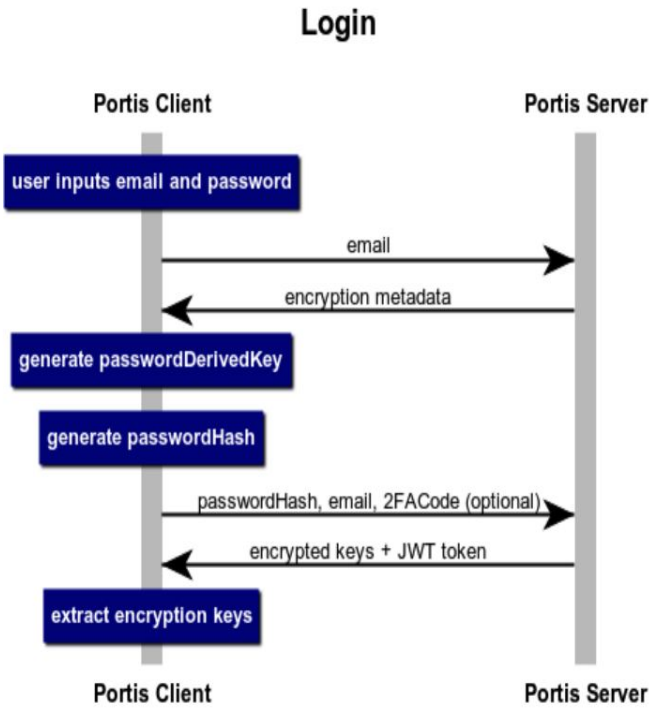
Step 2: The Client generates the passwordDerivedKey. The Client runs the

relevant Key Derivation Function with *password* as the secret, *email* as the salt and uses the relevant number of iterations obtained during Step 1. The Key Derivation Function returns a key, which we will call *passwordDerivedKey*.

Step 3: The Client generates a passwordHash. The Client runs PBKDF2 with SHA-256 as the pseudorandom function, using the *passwordDerivedKey* as the secret and the *password* as the salt, running a single iteration. This returns the *passwordHash*.

Step 4: The Client fetches encrypted keys and creates a session. The Client calls the relevant Server API, passing along the *email, passwordHash* and two-factor authentication code (if relevant). If the Server determines the credentials are valid, the response to the Client contains the *encryptedEncryptionKey* cipher and the *encryptedAsymmetricPrivateKey* cipher. In addition, the response contains an HttpOnly and secure cookie with a JWT token, which is used to maintain a standard session between the Client and the Server for subsequent API calls.

Step 5: Client extracts encryption keys. The Client runs the AES-CBC decryption algorithm, with the *encryptedEncryptionKey* as the ciphertext, the IV and the *passwordDerivedKey* as the secret, returning the plaintext *encryptionKey* . The Client runs the AES-CBC decryption algorithm, with the *encryptedAsymmetricPrivateKey* as the ciphertext, the IV and the *encryptionKey* as the secret, returning the plaintext *asymmetricPrivateKey* . The *asymmetricPrivateKey* is never stored in a persistent manner and only exists inside the browser's process memory, as a JavaScript variable.



3.3 Wallet Creation

To create a new wallet, the user must be signed in, which means the Client has the encryptionKey available in the browser's process memory and a valid JWT.

Step 1: The Client generates a blockchain wallet. The Client generates a wallet for the required blockchain using an appropriate library (for example, for Ethereum it uses the *createRandom* function of the ethers.js library). The newly created wallet will be composed of a *mnemonicPhrase* or other equivalent master seed phrase .

Step 2: The Client encrypts the wallet created in Step 1. The Client runs the AES-CBC encryption algorithm, with the wallet's *mnemonicPhrase* as the plaintext and the *encryptionKey* as the secret, returning the *encryptedMnemonicPhrase* cipher, composed of the *encryptedMnemonicPhrase*, the IV and the encryption algorithm mode.

Step 3: The Client stores the encryptedMnemonicPhrase cipher as well as the wallet public address on the Server.

3.4 Wallet Fetching

To fetch their wallet from the Server, the user must be signed in, which means the Client has the*encryptionKey* available in the browser's process memory and a valid JWT.

Step 1: The Client fetches the wallet. The Client calls the relevant Server API. If the JWT is valid, the response to the Client will contain the *encryptedMnemonicPhrase* cipher or the *encryptedPrivateKey* cipher (the latter in case the wallet was created via an imported private key, see **3.5 Wallet Import**).

Step 2: The Client decrypts the wallet. The Client runs the AES-CBC decryption algorithm, with the *encryptedMnemonicPhrase* or the *encryptedPrivateKey* as the ciphertext, the IV, and the *encryptionKey* as the secret, returning the plaintext *mnemonicPhrase* or *privateKey* .

3.5 Wallet Import

To import a wallet, the user must be signed in, which means the Client has the *encryptionKey* available in the browser's process memory and a valid JWT.

Step 1: The user inputs a mnemonic phrase or a private key.

Step 2: The Client encrypts the wallet provided by the user. The Client runs the AES-CBC encryption algorithm, with the *mnemonicPhrase* or *private key*

as the plaintext and the *encryptionKey* as the secret, returning the *encryptedMnemonicPhraseOrPK* cipher, composed of the *encryptedMnemonicPhrase* or *encryptedPrivateKey*, the IV and the encryption algorithm mode.
Step 3: The Client stores the *encryptedMnemonicPhrase* or the *encryptedPrivateKey* cipher as well as the wallet public address on the Server.

4 Persistent Session

The *encryptionKey* is never stored in a persistent manner and only exists inside the browser's process memory. That means that if a user resets their session while using a DApp (for instance: refreshes their browser), they will need to re-enter their credentials to access their wallet and sign transactions. From a UX perspective, this is an issue. To solve this problem, users locally store an encrypted copy of their credentials, which never leaves their device, using a secret stored on the Server. This secret can be fetched using a valid JWT, which expires after a set amount of time. The encrypted copy of their credentials, stored on their client, can only be decrypted with a secret stored on the Server, which can only be fetched using a valid JWT. As the JWT is an HttpOnly and secure cookie, it cannot be hijacked (for example via attack vectors such XSS), unless the attacker has direct physical access to the user's browser, assuming the device itself isn't compromised.

4.1 Persist Session

To persist their session, the user must be signed in, which means the following values are available on the Client: *email* ,*password*, *encryptionKey*, valid JWT.
Step 1: Get persistent session secret from the Server. The Client calls the Server API to fetch the persistent session secret. If the JWT is valid, the response to the Client will contain a *persistentSessionSecret* in the form of 32 random bytes. If a *persistentSessionSecret* already exists on the Server for the current JWT, that value is returned. Otherwise, a new *persistentSessionSecret* is generated on the Server and returned instead.
Step 2: Encrypt user credentials and store them in the Client. The Client runs the AES-CBC encryption algorithm, with a stringified JSON of the *email* and *password* as the plaintext and the *persistentSessionSecret* as the secret, returning the *encryptedCredentials* cipher, composed of the *encryptedCredentials*, the IV and the encryption algorithm mode. The *encryptedCredentials* cipher is stored in the browser's localStorage.

4.2 Use Persisted Session

To make use of a persisted session, the *encryptedCredentials* cipher must be stored in the Client.

Step 1: Verify the user logged into the DApp before. The Client checks if the DApp is authorized, i.e. if the user logged into this DApp before. If the DApp is not authorized the user will have to log in with their *email* and *password* instead of using the persistent session, even if the *encryptedCredentials* cipher is available on the Client. The Client identifies the DApp according to its domain, by inspecting the *document.referrer* value inside the Widget iframe. The Wallet is always considered an authorized DApp (even if the user's first login was to a DApp).

Step 2: Get persistent session secret from the Server and decrypt user credentials. If the *encryptedCredentials* cipher is available on the Client, and the DApp is authorized, the Client calls the relevant Server API to fetch the persistent session secret. If the JWT is valid, the response to the Client will contain a *persistentSessionSecret* in the form of 32 random bytes.

The client runs the AES-CBC decryption algorithm, with the *encryptedCredentials* as the ciphertext, the IV and the *persistentSessionSecret* as the secret, returning the plaintext *email* and *password* stringified JSON.

Step 3: Carry out the login flow. Once these two values are available in the Client memory, the Client can initiate the login flow. The login flow will, in turn, generate a new JWT, and as a result, a new *persistentSessionSecret* will be created in the persist session phase of the persistent session flow.

5 Two-Factor Authentication (2FA)

Two-factor Authentication works as an extra step in the login process, a second security layer, that will reconfirm the user's identity. Its purpose is to make attackers' life harder and reduce fraud risks. Portis uses otplib library, a Time-based (TOTP) and HMAC-based (HOTP) One-Time Password library[1], with a window parameter of 2 (meaning tokens in the 2 previous and the 2 future windows would be considered valid), and a time step of 30 seconds (meaning each window would last 30 seconds). We intend to add support for U2F at a later stage.

5.1 2FA Initialization

When a user sets up 2FA on their Portis account, their two-factor authentication secret is sent to the Portis Server, where it is encrypted with the AES GCM 256 encryption algorithm, using a secret stored on the Portis servers.

5.2 2FA Login flow

When the Client sends a login request with an *email* and *passwordHash*, although they are valid, the response could indicate that 2FA is enabled. In such a scenario, the Client will prompt the user to input the required 2FA code, and will then send the same request to the server, only this time with the 2FA code as well. If both the credentials and the 2FA code are valid, the response will be just like in the normal login flow, including the *encryptedEncryptionKey* cipher, the *encryptedAsymmetricPrivateKey* cipher, and the JWT cookie. From that point on the login flow can continue as normal.

6 Forgot Password

Other solutions which don't employ end-to-end encryption, store the private key on the user's device (either encrypted or plaintext), or alternatively - offer custodial services and store the user's private key on their server, letting the user interact with it through standard centralized authentication methods. The former solution means that if that user never backed up their private key and lost their device - they can never access their wallet again. The latter solution introduces the risk of lost or stolen private keys as the custodial service is a single point of failure which could be compromised. With Portis, as long as a user remembers their password, they can access their wallet from any device, and losing an existing device no longer means their crypto assets are gone forever. In addition, if the Portis servers were hacked and the encrypted private keys stolen, as long as the password chosen by the user is non-trivial (i.e. not susceptible to dictionary attacks), decrypting those private keys would be practically impossible. However, due to the nature of end-to-end encryption, if a user forgets their password, they also lose access to their private key, and hence to their crypto assets. To overcome this challenge, users are given a Backup Recovery Phrase, composed of 24 words, which they are encouraged to write down and store somewhere safe. There are two reasons for this mechanism, instead of telling users to write down and store a copy of their password. The first reason

is that the password they used might be sensitive and used for other services, so if their written-down Backup Recovery Phrase is compromised, it will not pose a risk to any of the other services for which the user is using that password. The second reason is to support our future plans of letting users store a copy of their encrypted private keys themselves (either on another device or using a different service, either centralized, like iCloud, or decentralized like IPFS). With the Backup Recovery Phrase and a copy of their encrypted private keys, users will be able to recover their private keys even if Portis ceases to exist.

6.1 Initialization

To restore a user's account in case they forget their password, the *passwordDerivedKey* is needed. That is because the *passwordDerivedKey* can decrypt the *encryptedEncryptionKey* stored on the Server, which in turn can decrypt the encrypted wallets also stored on the Server. In order to offer a more human-friendly solution for the user to backup and store that value (preferably by writing it down on a piece of paper), a Backup Recovery Phrase is generated on the Client, with the BIP39 *entropyToMnemonic* function, using the *passwordDerivedKey* as the entropy. This Backup Recovery Phrase, which consists of 24 words, is displayed to the user in the Client.

6.2 Recovery

To begin the recovery flow, the user inputs their *email* and Backup Recovery Phrase (24 words). The Client converts the Backup Recovery Phrase into the *passwordDerivedKey* using the BIP39 *mnemonicToEntropy* function. The Client hashes the *passwordDerivedKey* using SHA-512, returning a *passwordDerivedKeyHash* . The Client sends the *email* and *passwordDerivedKeyHash* to the Server, and if they match the values stored on the Server, the user receives an email with a random 6-digit code which was generated on the Server. This step adds an extra layer of security, as it verifies the user also owns the email they originally registered with, in addition to having the 24 words of the Backup Recovery Phrase.
The Create Account flow now takes place, meaning the user enters a new password into the Client and then sends the Server the following values: selected KDF, selected number of KDF iterations, 6-digit code, *email*, *passwordHash*, *oldPasswordDerivedKeyHash*, *newPasswordDerivedKeyHash*, *encryptedEncryptionKey* cipher, *asymmetricPublicKey* , and *encryptedAsymmetricPrivateKey* cipher. The Server returns to the Client the *oldEncryptedEncryptionKey*.

The client runs the AES-CBC decryption algorithm, with the *oldEncryptedEncryptionKey* as the ciphertext, the IV and the *oldPasswordDerivedKey* as the secret, returning the plaintext *oldEncryptionKey*.

The Login flow now takes place, with the new *password* . The encrypted wallets are fetched from the Server and for each wallet the client runs the AES-CBC decryption algorithm, with the *encryptedMnemonicPhrase* as the ciphertext, the IV, and the *oldEncryptionKey* as the secret, returning the plaintext *mnemonicPhrase* .

The wallet encryption phase of the Wallet Creation flow now takes place for each pre-existing wallet. The Client runs the AES-CBC encryption algorithm, with the wallet's *mnemonicPhrase* as the plaintext and the *newEncryptionKey* as the secret, returning the *encryptedMnemonicPhrase* cipher, composed of the *encryptedMnemonicPhrase* , the IV and the encryption algorithm mode.
The Client updates the encrytpedMnemonicPhrase cipher on the Server.

7 Summary

The average person won't and shouldn't care about blockchain. To promote mainstream adoption we need to make sure we abstract away from the user as much as we can without compromising security. Most importantly, that means that said abstraction cannot extend to the realm of their private keys.

Private key management is one of (if not) the biggest challenges for users in the blockchain ecosystem.

While custodial solutions are able to offer users an experience which feels very familiar, we believe they are a step in the wrong direction, as "unstoppable apps" are not really unstoppable if they hinge on a centralized service, as they could be compromised in various ways. Alternatively, placing the burden squarely on the shoulders of users by requiring them to manage their private keys on their own is too demanding of individuals who are not necessarily tech-savvy. By using end-to-end encryption, users enjoy a familiar experience (email and password login flow), without requiring them to install anything, while making sure they are the only ones able to access their unencrypted private keys. In addition, accessing the same private key on multiple devices is as simple as typing in their credentials in their new device. In addition, just as the common user doesn't care about blockchain, they definitely don't care *which* blockchain. For that reason, Portis supports mul-

tiple blockchains, which makes life easier for both users and developers. Finally, we are focusing on numerous usability issues, to make sure any transition from a standard web2 to a decentralized application will be as frictionless as possible. Such efforts include integrating with the Gas Stations Network[2], to remove the burden of network fees from users, offering direct-to-wallet purchases of cryptocurrency through services such as sendwyre.com, and working together with DApp developers to better understand and tackle useflow pain points.

8 What's Next?

Our long term vision is to let users manage multiple private keys for different types of DApps, encrypting each of those keys differently. Each of those keys will have varying roles in a single identity smart contract, which will provide additional levels of control and account recovery mechanisms. So while a user might want your end-to-end encrypted private key to have the highest permissions level, they can also generate unencrypted private keys for interacting with DApps that have little to no value. Obviously, there will also be a migration path for the private keys used for DApps. A user might start off by interacting with a DApp using an unencrypted private key, but will revoke that key and assign their end-to-end encrypted key as the one permissioned to interact with that DApp at a later stage, as their account accumulates value in said DApp.

References

[1] Time-based (TOTP) and HMAC-based (HOTP) One-Time Password library, https://github.com/yeojz/otplib

[2] EIP-1613, Gas Stations Network, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1613.md, Yoav Weiss, Dror Tirosh, Alex Forshtat, Tabookey, 2018