Permissioned Web2<>Web3 Account Design Doc

Background

There is a web portal where users sign in using their email and password combination. The public, private key pair is created on the backend and custodied by the system - that is, the user didn’t have a wallet before-hand & doesn’t know what their private key is. The wallet functionality is limited to only signing off on multi-sig, and does not include permissions to initiate transactions - such as token transfers or NFT minting. Assume this user belongs to a large organisation like a bank. To make user experience simple, we wouldn’t want every single user from the bank interacting with the blockchain to manage gas but rather we would want it managed at an entity level.

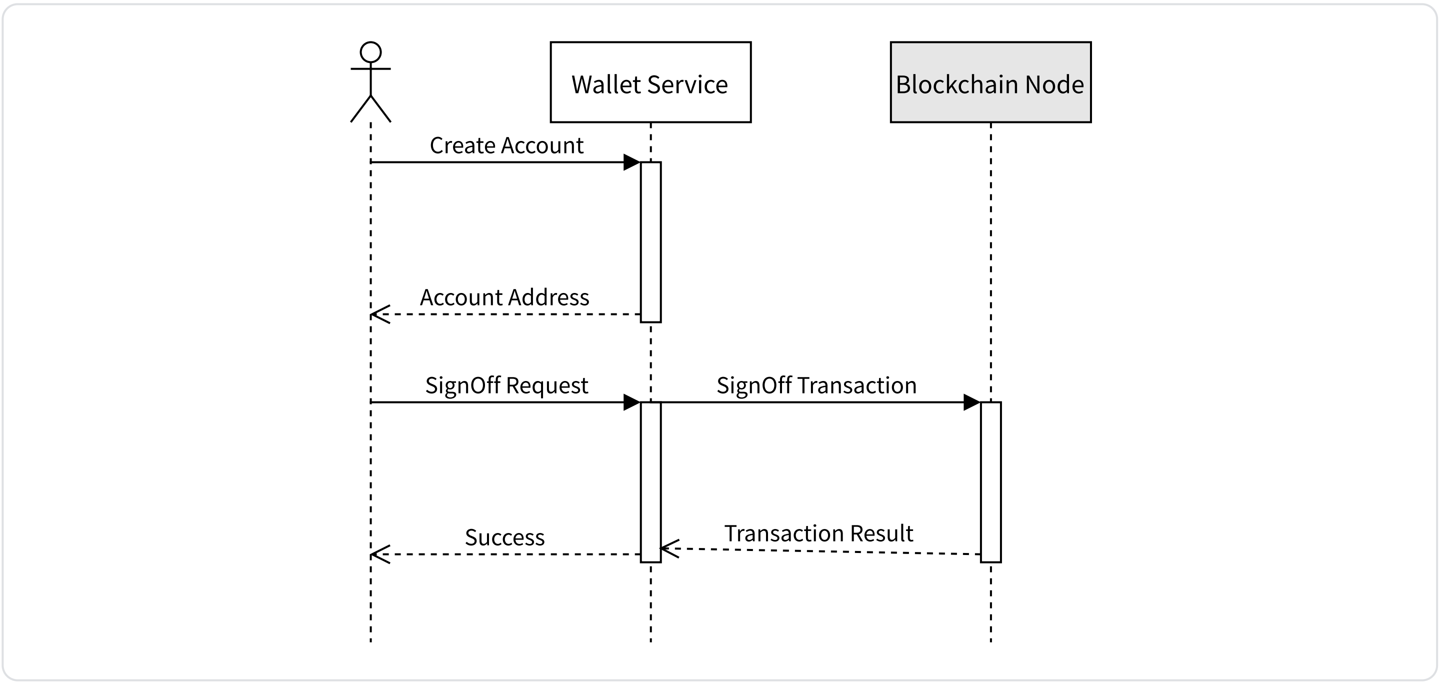
Requirements

We need to provide one wallet service with the following functions:

1. Account Creation
   When account creation request is received, new private key、public key and blockchain address should be generated and saved properly. The blockchain address should be returned to service requester but the private key should be kept safely and privately.

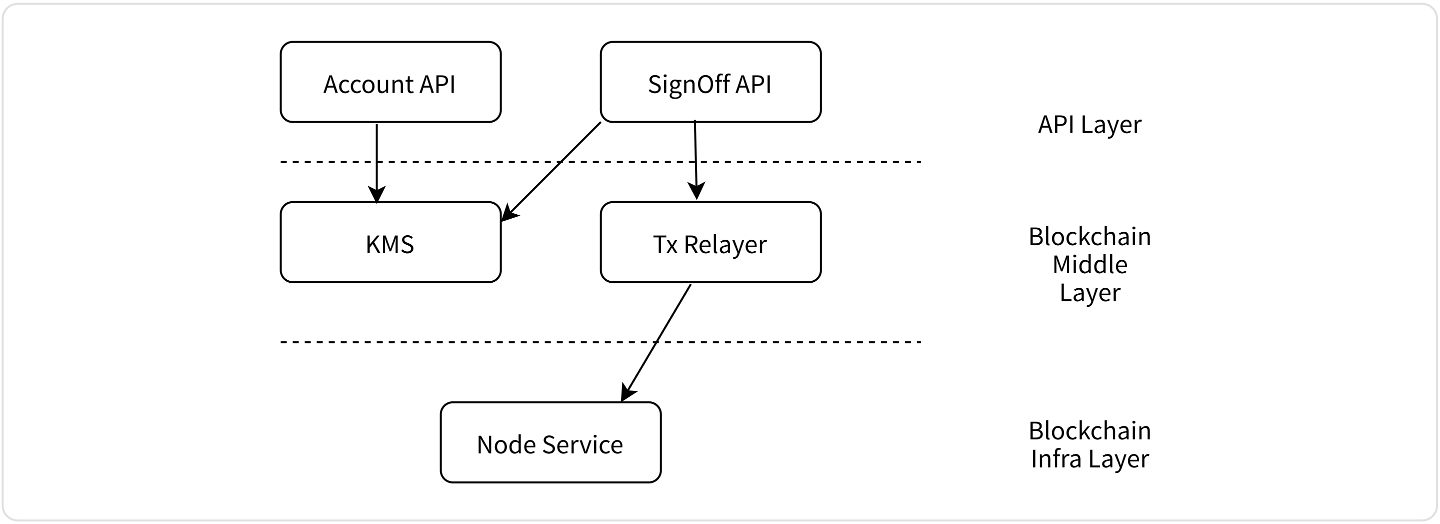
2. Multi-sig sign off
   This is the only business function supported by wallet service for now. But the design should be scalable enough that further business functions could be added easily.
   The sign off request from user should be signed with associated private key. The signature should be saved and could be easily verified.
   The sign off request should be sent to blockchain, but the user doesn't need to care about gas fees.

Below diagram could demonstrate the requirements in high level.



Technical Solution

High Level Architecture



We could split the system into three layers at a high level:

API Layer: It's the interface interacting with UI or other systems.
1. Account API
It provides the rest api to CRUD account info. The account is identified by email but every account should have one unique id and blockchain address. This component calls KMS to generate blockchain address.

2. SignOff API
It provides the rest api to accept signoff requests. It constructs the signoff request payload based on MultiSig smart contract ABI and then calls KMS to sign the payload and finally sends the signed payload to Tx Relayer.

Blockchain Middle Layer: It contains components that could be shared with different businesses besides "SignOff".
1. KMS: Key Management Service.
The private key and blockchain address are generated and maintained here. We could leverage 3rd party services such AWS KMS to maintain the data or keep the data in our own database.
Security is the key to this component:
- Private key should be generated randomly enough.
- Generated private keys should be kept securely. If it's kept in our own database, it should be encrypted with AES algorithm and the AES master key should be different for each private key.
- Strict limited access rule should be applied to this component.

Since the private key is kept here. It also provides a service to sign data.

2. Tx Relayer: The user signoff request is sent to blockchain node by this component.

It mainly contains three parts: **Entity level blockchain account**, **TxRelayer Smart Contract** and **TxRelayer backend service**.
When TxRelayer backend service accepts the signed request payload, it will encapsulate the payload to construct and sign the transaction to invoke TxRelayer Smart Contract, the TxRelayer Smart Contract will validate the user signed request payload and forward the request to MultiSig smart contract.

Entity level blockchain account pays the gas for blockchain transactions, so that we should always keep enough funds in this account.

Blockchain Infra Layer: It provides reliable node services to interact with blockchain networks.

Drawback of above solution

Since all the transactions are sent via TxRelayer Smart Contract, the MultiSig smart contract or other business smart contracts have to follow https://eips.ethereum.org/EIPS/eip-2771 to recognize the original user address who sends the signoff request.

To avoid the drawback, we could create one Smart Account for each user. The Smart Account basically is one smart contract like below.

```
contract SmartAccount {
    address userAddress;
    address entityOperator;

    function verifyAndSend(address target, bytes memory requestData, bytes signa
        // verify the request is sent from predefined userAddress
        // call target business smart contract
    }
    ......
}
```

The entityOperator could send transactions on behalf of the user and the user's SmartAccount address is recognized as msg sender in target business smart contract.

Technical Details

1. What functions will you use to create keys?

The private key is fundamentally one random number. We should use crypto package to generate the random number no matter what language we use.

If we choose golang, we could import go-ethereum cryoto package to generate private key with below codes:

```
privateKey, err := crypto.GenerateKey()
if err != nil {
  log.Fatal(err)
}
privateKeyBytes := crypto.FromECDSA(privateKey)
fmt.Println(hexutil.Encode(privateKeyBytes)[2:])
```

2. How will you keep the private keys safe?

A. Keep the private keys in standalone hardware with strictly limited access to internet.

B. All keys are stored with strong encryption. Never expose private keys in plaintext. Only decrypt the encrypted keys when data signing is required.

3. What edge cases may exist in this process?

A. It's possible to get the same signoff request many times. Gas fees are wasted if we handle all the requests with normal processes. We should be able to identify which ones are normal ones and which ones are from replay attack.
Before sending the request, one "nonce" value should be generated and signed together with signoff data. The request with duplicated "nonce" value will be ignored.

B. Each entity level account can only handle requests one by one. More entity level accounts should be configured if we want to handle more requests at a time.

C. All blockchain transactions are handled asynnously. That means we can not know the transaction result after sending out the transactions. We need to monitor the blockchain to get the result.
D. It's possible for some reason that the transaction is not sent out properly. We should properly handle the fail cases and ensure the proper transaction is sent out successfully.