Simple Summary

A new wallet paradigm that separates a user's main account, where valuable assets are stored, from "leaf accounts", embodied in a Leaf Wallet framework, that are used for day-to-day transactions. Think of leaf wallets as prepaid cards. This system significantly reduces the risk of losing access to assets and provides a more seamless and secure way of interacting with dApps on the Ethereum blockchain. Furthermore it is super easy to implement.

Abstract

The Leaf Wallet paradigm introduces a new way of managing EOA accounts and assets on the Ethereum blockchain. It works by separating a user's main wallet, a more technically definition is main account, which holds their valuable assets, from "leaf wallets", a more technically definition is leaf accounts, that are used for everyday transactions. The user typically manages the main account in a trusted environment, whereas the leaf wallet might be managed on smart devices. Even smart watches and similar. This separation allows users to interact with dApps on these devices and make transactions without exposing their main wallet to potential security risks. In the event that a leaf wallet is lost or compromised, the main wallet can still access and control the assets assigned to the lost leaf wallet. Furthermore, users can easily generate new leaf wallets and register them with their main wallet, block access to leaf wallets and delegate access to a new leaf wallet, allowing for ease of recovery and delegation.

Motivation

The motivation behind the Leaf Wallet paradigm is to address some of the common issues faced by blockchain wallet users such as the risk of losing access to their assets due to loss of private keys or seed phrases and having the need to use third party wallet managers like Metmask or wallet integrators like Wallet Connect. With traditional blockchain wallets, losing private keys or seed phrases often means losing access to assets since these are the only ways to unlock and gain control over a blockchain wallet. By separating the main wallet from leaf wallets, the Leaf Wallet paradigm significantly reduces this risk and provides a more secure and user-friendly way of interacting with dApps and the Ethereum blockchain. Further removing the need of centralized services like Wallet Connect or Metamask promotes decentralization and ease of adoption.

Core Building Blocks

Leaf-Wallet Enabled dApp

The 'Leaf-Wallet Enabled dApp' forms the user-facing layer of the Leaf Wallet Paradigm. These dApps are designed to allow users to interact securely using their leaf wallets, thereby reducing the risk exposure of their main wallet. They provide an interface for daily transactions and interactions, acting as an intermediary between the user's main wallet and leaf wallets. The dApp also takes care of generating and storing the leaf wallets securely on the user's device and verifying the successful registration and activation of the leaf wallets.

Registry Enabled dApp Smart Contract

The 'Registry Enabled dApp Smart Contract' is another integral part of this paradigm. This smart contract handles the standard functionalities of the dApp, but also includes specific functions that allow for the creation, management, and activation of leaf wallets. Implementing the 'Registry Contract' functionality, it provides a mechanism for registering leaf wallets to a user's main wallet on the Ethereum blockchain. This enables the main wallet to maintain control over leaf wallets and provides an additional layer of security. Additionally, it also ensures a unique activation code is generated and stored during the registration process, which is used for confirming the correct assignment of a leaf wallet to the main account.

dApp Leaf-Wallet Registration App

Finally, the 'dApp Leaf-Wallet Registration App' is an essential tool that interacts directly with the user's main wallet. It performs several critical functions, from requesting the QR code of a leaf wallet for registration purposes to executing registry contract functions. Additionally, it presents the activation code to the user as part of the leaf wallet activation process. This web app also facilitates the transfer of minimal funds from the main wallet to the leaf wallet. These funds are essential to cover the gas costs required for transactions. By doing so, this application ensures a seamless and secure registration and management process for leaf wallets under the control of a user's main wallet.

Specification

Main Wallet and Leaf Wallets

In the Leaf Wallet paradigm, a user's main wallet is kept secure in a trusted environment and is used to store valuable assets. Leaf wallets, on the other hand, are used for everyday transactions such as interacting with dApps.

dApp Registry Smart Contract

Each leaf wallet is registered to the main wallet via a registry contract. This means that all assets assigned to a leaf wallet are also assigned to the main wallet. Thus, even if a user loses access to a leaf wallet, they can still access and control their assets through the main wallet.

Developers following the Leaf Wallet paradigm need to implement the following minimum interface within their smart contract, ensuring compatibility with the Leaf Wallet functionality. Below is an abstracted interface and its subsequent full contract implementation.

Abstracted Interface

```
interface ILeafWalletRegistry {
    function isSenderRegistered() external view returns (bool);
    function assignAddressToSender(address assignee)  external returns
(uint256);
    function getRelatedDevices() external view returns (address[] memory);
    function activateDevice(uint256 activationCode) external;
    function isDeviceActivated(address leafWallet) external view returns
(bool);
}
```

This interface contains the minimum required functions to allow for the registration and management of leaf accounts. Here's what each function does:

isSenderRegistered: This function checks if the current sender (i.e., leaf account) is already registered in the registry contract.

assignAddressToSender: This function allows the main account to assign a new leaf account. It returns a unique activation code

getRelatedDevices: This function enables the main account to retrieve a list of all associated leaf accounts.

activateDevice: This function is called by a leaf account to mark itself as active. It consumes the activation code the user provides based on the code shown in the registration app. This way only if the leaf account is assigned to the correct owner the leaf account can successfully execute this function. This adds an extra layer of security by ensuring that only leaf accounts generated in a secure environment on the user's device can be activated. The leaf account needs to have minimal funds transferred from the main account to pay for the gas required for this operation.

isDeviceActivated: This function allows any entity to check if a specific leaf account has been activated. It is useful for verifying that the leaf account has been correctly initialized and is ready for regular transactions.

Base Registry Contract Implementation

```
contract LeafWalletRegistry is ILeafWalletRegistry {
    mapping(address => address) private _deviceOwner;
    mapping(address => address[]) private _ownerDevices;
    mapping(address => bool) private _deviceActivated;
    mapping(address => uint256) public _deviceActivationCodes;

    function isSenderRegistered() public view override returns (bool) {
        return _deviceOwner[msg.sender] != address(0);
    }

    function assignAddressToSender(address assignee) public override returns (uint256){
        require(_deviceOwner[assignee] == address(0),"Device already registered");

        _ownerDevices[msg.sender].push(assignee);
        _deviceOwner[assignee] = msg.sender;

        uint256 activationCode = uint256(keccak256(abi.encodePacked(block.timestamp,
block.difficulty, msg.sender))) % 10000;
        _deviceActivationCodes[assignee] = activationCode;

        return activationCode;
    }

    function getRelatedDevices() public view override returns (address[] memory) {
        return _ownerDevices[msg.sender];
    }

    function activateDevice(uint256 activationCode) external{
        require(_deviceOwner[msg.sender] != address(0),"Device not yet registered");
        require(_deviceActivationCodes[msg.sender] == activationCode, "Activation code does not
match.");

        _deviceActivated[msg.sender]=true;
    }

    function isDeviceActivated(address leafWallet) external view returns (bool){
        return _deviceActivated[leafWallet];
    }

}
```

This contract adheres to the **ILeafWalletRegistry** interface, implementing the necessary functionality for registering and managing leaf accounts associated with the main account. Note that data storage operations have been omitted from this example for simplicity. If needed, the contract can be expanded to include additional functionalities as per the application requirements.

Developers working on the Leaf Wallet paradigm are provided with a baseline contract **LeafWalletRegistry**. They can either directly extend this contract, or they can choose to implement their own custom contract based on the provided interface **ILeafWalletRegistry**.

Here's the key point to remember: regardless of whether you choose to extend the provided contract or implement your own, your contract should adhere to the **ILeafWalletRegistry** interface. This is necessary to ensure compatibility and seamless interaction with the Leaf Wallet functionality.Though it is not mandatory to use this interface to follow the Leaf Wallet Paradigm.

Extending the Provided Base Contract

If the provided base contract meets your application's requirements, or you only need minor modifications, it may be most efficient to extend the LeafWalletRegistry contract directly. This allows you to inherit all its functionality, which you can then build upon or modify as needed.

```
pragma solidity ^0.8.0;

import "./LeafWalletRegistry.sol";

contract MyLeafWalletRegistry is LeafWalletRegistry {
    // Additional functionalities or modifications go here
}
```

Implementing Your Own Contract

Alternatively, you may find that the base contract does not quite suit your needs. In this case, you can write your own contract from scratch. However, to maintain compatibility with the Leaf Wallet functionality, your contract should implement the ILeafWalletRegistry interface.

```
pragma solidity ^0.8.0;

import "./ILeafWalletRegistry.sol";

contract MyLeafWalletRegistry is ILeafWalletRegistry {
    // Implement the functions defined in ILeafWalletRegistry here
    // Add any additional functionalities needed for your
application
}
```

With either approach, the key is to at best adhere to the ILeafWalletRegistry interface, ensuring that your contract implements the isSenderRegistered, assignAddressToSender, and getRelatedDevices functions. This consistency allows for seamless interaction with dApps following the Leaf Wallet paradigm, regardless of the specific implementation details of your contract.

Leaf Wallet enabled dApp Implementation

When implementing the Leaf Wallet paradigm within a dApp, developers can follow this specific flow of actions to establish the leaf account:

1. Check for Leaf Account Existence: Once the user starts the dApp following the Leaf Wallet paradigm on their device, the first task for the dApp is to check if a leaf account already exists on the user's device. This can be achieved by checking the local storage or secure storage of the user's device for a leaf account's private key.
2. Create New Leaf Account: If the check in the first step concludes that no leaf account exists, the dApp should then generate a new unique leaf account for the user. The leaf account is created using cryptographic algorithms, resulting in a unique pair of public and private keys. The private key is then stored securely on the user's device.
3. Check if Leaf Account is Registered: After ensuring that a leaf account exists on the device, whether pre-existing or newly created, the dApp must then verify if the leaf account is registered with the user's main account. The dApp should execute a check against the registry contract on the blockchain, using the public key of the leaf account.
4. Provide Public Key as QR Code: If the leaf account is not already registered, the dApp should display a QR code containing the public key of the leaf account. The user can then scan this QR code in a trusted environment using the Registration feature of the dApp to link the leaf account with their main account.
5. The dApp then monitors the registration state and after registration verification provides the activation code to the user:
6. Compare Verification Code: The user is then asked to compare the code shown in the dApp with the one shown in the Registration App.
7. If the user verifies the activation code successfully he activates the leaf wallet in the smart contract. Note that sufficient funds must be available in the leaf account for this step to succeed.

Sample implementation based on library @leaf-wallet/device

```
async initialize(password) {
    // Stage 1: Initializing Environment
    // Wait until the library is ready
    while(!this.leafwalletdevice.isReady) {
        await timeout(100);
    }
```

```
    // Stage 2: Confirming dApp Wallet
    // Check if a wallet already exists in the secret storage
    if(!await this.leafwalletdevice.checkWalletExists(password)) {

        // Stage 3: Creating dApp Wallet
        // If no wallet exists, create a new one
        await this.leafwalletdevice.createWallet(password);
    }

    // Stage 4: Loading dApp Wallet
    // Load the wallet from the secret storage
    this.wallet = await this.leafwalletmobile.loadWallet(password);

    // Stage 5: Verifying dApp Registry
    // If the wallet is registered, we're done. If not, move to registration
stage
    if(!await this.leafwalletdevice.isRegistered()) {
        return false;
    }

    // Stage 6: Checking Activation Status
    // If the wallet is not activated, trigger activation
    if(!await this.leafwalletdevice.isActivated()) {

      // update UI with activation code
      const activationCode = await this.leafwalletdevice.getActivationCode();

      //wait for user confirmation
      await activationCodeConfirmation();
      // Stage 7: Activating dApp Wallet
      // Trigger activation of the leaf wallet with the activation code
      await this.leafwalletdevice.activateDevice(activationCode);
    }

    // If the wallet is registered and activated, we're done
    return true;
}
```

This flow ensures that each leaf account is properly created, checked, and registered, leading to a secure and efficient user experience when interacting with dApps utilizing the Leaf Wallet paradigm.

Security Considerations for Leaf Wallet enabled dApps

The security measures for the dApp leaf wallet implementation vary depending on the value of the assets created by the dApp and the risk of compromise. The developer has the option to decide between the following security levels depending on the deployment context or even go far beyond these:

a) Browser-based dApp with Static Password
The leaf wallet private key is encrypted with a static password and saved in IndexedDB or a similar browser-based web context segregated storage system. This approach provides a balance between convenience and security. However, it might be more susceptible to security risks as the static password, if exposed, could compromise the leaf wallet.

b) Browser-based dApp with User-defined Password
The leaf wallet private key is encrypted with a user-defined password and saved in IndexedDB or similar. This approach gives the user more control over their security by allowing them to set a unique, perhaps stronger password. However, it also means that the user must remember their password, as forgetting it could result in losing access to their leaf wallet. Also it would require the user to enter the password at least at each dApp startup.

c) Mobile Native dApp with Device Key Store
The leaf wallet private key is stored in the device's secure key store. This approach is typically more secure as it leverages the device's built-in security features. However, it might be more complex to implement and may limit the dApp's compatibility to only devices that support secure key storage.

d) Browser-based dApp with Web Authentication API
Web Authentication API (WebAuthn) is a web standard published by the World Wide Web Consortium (W3C). It provides a way to securely create and use public key-based credentials for the purpose of strongly authenticating users. When applied to the Leaf-Wallet Paradigm, the Leaf Wallet private key isn't stored anywhere in the application or user device. Instead, the key pair is securely created and stored within an authenticator (like a security key, biometrics-enabled device, etc.) using the WebAuthn API. The private key never leaves the authenticator.

This approach leverages the strong security of public key cryptography and the convenience of the authenticator, which could be something the user already has, like a fingerprint scanner on their device or a USB security key.

While this option potentially provides the highest security level, it also requires the most effort to implement and might introduce usability challenges. The user needs to have a

compatible authenticator, and the development team needs to carefully design the user experience around the authenticator to maintain convenience and ease of use.

In any way, it's recommended if feasible to implement always security measures like two-factor authentication or biometric authentication to further protect the leaf wallet.

Asset Management in the Contract

In the Leaf Wallet paradigm, assets are either never or also bound to the leaf wallet but are always tied to the main wallet registered with the dApp. This is done by assigning all assets of a leaf wallet to the main wallet via the registry contract.

Here is a simple implementation of an ERC-721 contract adapted to the Leaf Wallet paradigm, using OpenZeppelin contracts as a starting point:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "./ILeafWalletRegistry.sol";

contract MyLeafWalletERC721 is ERC721 {
    ILeafWalletRegistry private _leafWalletRegistry;

    constructor(ILeafWalletRegistry leafWalletRegistry)
ERC721("MyLeafWalletERC721", "MLWE") {
        _leafWalletRegistry = leafWalletRegistry;
    }

    function _isApprovedOrOwner(address spender, uint256 tokenId)
internal view override returns (bool) {
        address owner = ownerOf(tokenId);
        address mainAccount =
_leafWalletRegistry.getMainAccount(owner);

        // Approval check is extended to include the main account
        return (spender == owner || spender == mainAccount ||
getApproved(tokenId) == spender  || getApproved(tokenId) ==
mainAccount || isApprovedForAll(owner, spender) ||
isApprovedForAll(mainAccount , spender));
    }

    function mint(address to, uint256 tokenId) public {
        _mint(to, tokenId);
```

```
        }
}
```

In the above contract, we override the _isApprovedOrOwner function, which is used in the transferFrom function of the ERC-721 standard to check if the msg.sender is allowed to transfer the token. We add a condition to check if the msg.sender is the main account linked with the token owner. This way, the main account can also transfer the token.

This contract also assumes you have a getMainAccount function in your ILeafWalletRegistry interface, which returns the main account associated with a given leaf account. If your registry contract does not have such a function, you will need to implement it.

Registration Web App

1. Main Wallet Connection

1.1 Purpose
Allow users to connect their main wallets to the Registration Web App.

1.2 Specification
The application should provide an interface to connect to the user's main wallet. This can be implemented using wallet connection protocols such as Web3 or WalletConnect.

2. Leaf Wallet QR Code Request

2.1 Purpose
Retrieve the public key of the leaf wallet the user wishes to register.

2.2 Specification
The application should allow the user to scan or input the QR code of the leaf wallet. This QR code is generated during the creation of the leaf wallet and contains the public key of the leaf account.

3. Registry Contract Function Execution

3.1 Purpose
Link the main wallet and leaf wallet within the Ethereum blockchain via the registry contract.

3.2 Specification
Upon obtaining the leaf wallet's public key, the application should execute the registry contract functions to register the leaf wallet under the user's main wallet by calling the assignAddressToSender function in the smart contract.

Generate Verification Code: In response, the smart contract generates a unique random number (or code) and stores it on the blockchain. This code is linked with the leaf wallet's address.

4. Verification

4.1 Purpose
Confirm that the leaf wallet registration was successful and the dApp is now fully functional.

4.2 Specification
The Registration Web App should provide feedback indicating successful registration of the leaf wallet with the main wallet. Additionally display verification code: The generated code is then displayed in the Registration App for the user to compare to the code shown in his dApp.

5. Transfer Minimal Funds

5.1 Purpose
Ensure that the leaf wallet has sufficient funds to pay for gas costs associated with future transactions.

5.2 Specification
Following successful verification, the application should facilitate the transfer of a minimal amount of Ether (or other gas-paying token) from the main wallet to the leaf wallet. This amount should be sufficient to cover the gas costs for the anticipated number of transactions.

This method introduces a human verification step, which would make it practically impossible for an attacker to front-run this process successfully. The only way for an attacker to successfully manipulate this process would be if they could somehow predict the random number generated by the smart contract, which would be computationally infeasible if a sufficiently strong random number generation method is used.

Security Considerations

While designing and implementing the Leaf Wallet Paradigm, certain security aspects need to be taken into account to ensure the safety of the user's assets. One possible vulnerability that may emerge relates to the registration leaf wallet public keys.

Front-Running Attack: A front-running attack is a potential risk that needs to be mitigated. This type of attack occurs when an adversary attempts to take advantage of the knowledge of a pending transaction by executing their own transaction first, thereby possibly altering the

outcome of the original transaction. For instance, a malicious entity could monitor the Ethereum network for pending leaf wallet registration transactions, duplicate the public keys from these transactions, and attempt to assign these keys to their own main wallet. If successful, this could result in the hijacking of the leaf wallet before the legitimate user completes their registration process. To prevent such attacks, the implementation of an activation step with a shared secret between registration app and dApp, that only the leaf wallet can successfully execute, is crucial. It verifies the registration and locks the leaf wallet's association with the main wallet. This additional layer of security ensures that even if a malicious actor tries to front-run the leaf wallet registration, they will be unable to take control of the leaf wallet due to the activation step that can only be executed by the legitimate leaf wallet itself."

Rationale

The Leaf Wallet paradigm offers several key benefits:

1. **Improved Security**: By separating the main wallet from leaf wallets, the Leaf Wallet paradigm reduces the risk of losing access to assets due to loss of private keys or seed phrases. Even if a leaf wallet is lost or compromised, the main wallet can still access and control the assets.

2. **Ease of Recovery**: If a leaf wallet is lost or compromised, a user can simply generate a new leaf wallet and register it with their main wallet. The lost leaf wallet can be safely disregarded.

3. **Enhanced User Experience**: The Leaf Wallet paradigm provides a more seamless and secure way of interacting with dApps and the Ethereum blockchain, enhancing the overall user experience.

4. **Fully Decentralized**: Unlike other solutions that rely on centralized APIs, Leaf Wallet operates entirely on-chain and doesn't depend on any centralized infrastructure, ensuring that interactions with dApps remain decentralized

5. **Future Proof**: As Ethereum continues to innovate, with changes like Ethereum 2.0 and account abstraction on the horizon, adopting the Leaf Wallet paradigm ensures that your dApp or wallet is ready to adapt to these future changes.

The prospect of Ethereum moving towards account abstraction provides an ideal backdrop for the implementation of the Leaf Wallet paradigm. It is a forward-thinking approach that prioritizes user experience and security while providing a flexible, future-proof model for Ethereum interactions.