Blume Liquid Staking

Staking Contract

BLS Token

stBLS Token

---

1. Introduction

The **Blume Liquid Staking Contract** allows users to stake BLS tokens and mint stBLS tokens, representing their staked balance. It provides a secure mechanism to manage staking and unstaking operations while maintaining transparency and control over the process.

2. Features

- **Token Management**:
  - Users can stake BLS tokens to receive an equal amount of stBLS tokens.
  - stBLS tokens represent ownership of the staked BLS tokens and are burned upon unstaking.
- **Secure Staking**:
  - Implements strict access control where only the staking contract can mint or burn stBLS tokens.
- **Transparency**:
  - Emits events for staking and unstaking actions to ensure all transactions are auditable.

3. Architecture

- **Contracts**:
  - **BLSToken**: ERC20 token representing the native token, BLS.
  - **StakedBLSToken**: ERC20 derivative token representing staked tokens (stBLS).
  - **BlumeLiquidStaking**: Manages staking, unstaking, and tracking user balances.
- **Key Relationships**:
  - The BlumeLiquidStaking contract interacts with:
    - **BLSToken**: Transfers BLS tokens between users and the contract.
    - **StakedBLSToken**: Mints and burns stBLS tokens during staking and unstaking.

4. Contract Details

State Variables

- BLSToken public blsToken: Reference to the BLS token contract.
- StakedBLSToken public stakedBlsToken: Reference to the stBLS token contract.
- uint256 public totalStaked: Tracks the total amount of staked BLS tokens.
- mapping(address => uint256) public stakedBalances: Tracks each user's staked balance.

Functions

1. Constructor

constructor(BLSToken _blsToken, StakedBLSToken _stakedBlsToken) { ... }

- Initializes the contract by linking BLSToken and StakedBLSToken addresses.

2. Stake

function stake(uint256 amount) external { ... }

- **Description**: Allows users to stake BLS tokens and receive stBLS tokens.
- **Logic**:
  - Transfers BLS tokens from the user to the contract.
  - Mints stBLS tokens for the user.
  - Updates totalStaked and the user's stakedBalances.
- **Events**: Emits Staked with user address and amount.

3. Unstake

function unstake(uint256 amount) external { ... }

- **Description**: Allows users to unstake stBLS tokens and redeem an equivalent amount of BLS tokens.
- **Logic**:
  - Burns stBLS tokens from the user's account.
  - Transfers BLS tokens back to the user.
  - Updates totalStaked and the user's stakedBalances.
- **Events**: Emits Unstaked with user address and amount.

Events

- event Staked(address indexed user, uint256 amount): Logs when a user stakes tokens.
- event Unstaked(address indexed user, uint256 amount): Logs when a user unstakes tokens.

5. Deployment Guide

Dependencies

- The contract requires pre-deployed instances of:
  - BLSToken (ERC20 token contract).
  - StakedBLSToken (ERC20 token contract).

Deployment Steps

1. Deploy BLSToken and StakedBLSToken contracts.
2. Deploy BlumeLiquidStaking by passing the addresses of the deployed BLSToken and StakedBLSToken contracts.
3. Set the staking contract address in StakedBLSToken using the setStakingContract function.

6. Testing Scenarios

1. Stake Functionality

- **Input**: User stakes 100 BLS.
- **Expected Outcome**:
  - User's BLS balance decreases by 100.
  - User receives 100 stBLS.
  - totalStaked and stakedBalances reflect the staked amount.

2. Unstake Functionality

- **Input**: User unstakes 50 stBLS.
- **Expected Outcome**:
  - User's stBLS balance decreases by 50.
  - User receives 50 BLS.
  - totalStaked and stakedBalances reflect the reduced amount.

3. Minting and Burning Validation

- **Test Case**: Ensure only BlumeLiquidStaking can mint and burn stBLS.
- **Expected Outcome**: Any unauthorized call to mint or burn should fail.

7. Security Considerations

- **Access Control**:
  - Only BlumeLiquidStaking can mint or burn stBLS tokens.
- **Validation**:
  - require statements ensure valid inputs for staking and unstaking.
  - Prevents staking or unstaking with zero amounts.

8. Future Improvements

- Add staking rewards to incentivize users.
- Implement time-based locks for staking periods.
- Integrate governance mechanisms for staked tokens.

Smart Wallet

SmartWallet Contract

---

1. Introduction

The **Ethereum Smart Wallet** provides users with a secure and convenient way to manage their funds on the Ethereum blockchain. The wallet integrates multiple authentication methods, such as social media OTPs, biometric verification, and passcodes, to ensure user security and ease of access.

2. Features

- **Multifactor Authentication**:
  - **Passcode**: Users can set a passcode hashed on-chain for wallet access.
  - **OTP Verification**: Off-chain service integration allows users to verify OTPs before transactions.
  - **Biometric Verification**: Off-chain service integration enables biometric authentication.
- **Fund Management**:
  - Deposit Ether securely.
  - Withdraw Ether only after successful multifactor authentication.
- **Security**:
  - Requires all authentication methods (Passcode, OTP, Biometric) for withdrawals.
  - Resets OTP and biometric verification after each transaction.

3. Architecture

Contracts:

- **SmartWallet**: The main contract that handles user deposits, withdrawals, and authentication verification.

Key Features:

- Stores user passcodes securely using a hashed format on-chain.
- Maintains OTP and biometric verification statuses for each user.
- Implements event logging for all key wallet actions.

4. Contract Details

State Variables

- mapping(address => bytes32) private passcodeHashes: Stores hashed passcodes for each user.
- mapping(address => bool) private otpVerified: Tracks OTP verification status for each user.
- mapping(address => bool) private biometricVerified: Tracks biometric verification status for each user.

Functions

1. Deposit

function deposit() external payable { ... }

- **Description**: Allows users to deposit Ether into the wallet.
- **Logic**: Ensures the deposit amount is greater than zero.
- **Event Emitted**: FundsDeposited(address user, uint256 amount).

2. Set Passcode

function setPasscode(bytes32 passcodeHash) external { ... }

- **Description**: Allows users to set or update their passcode.
- **Logic**: Stores the hashed passcode securely on-chain.
- **Event Emitted**: PasscodeSet(address user, bytes32 passcodeHash).

3. Verify OTP

function verifyOTP() external { ... }

- **Description**: Updates the OTP verification status after successful off-chain verification.
- **Logic**: Sets the user's otpVerified status to true.
- **Event Emitted**: OTPVerified(address user).

4. Verify Biometric

function verifyBiometric() external { ... }

- **Description**: Updates the biometric verification status after successful off-chain verification.

- **Logic**: Sets the user's biometricVerified status to true.
- **Event Emitted**: BiometricVerified(address user).

5. Withdraw

function withdraw(uint256 amount, string memory passcode) external { ... }

- **Description**: Allows users to withdraw Ether after successful authentication.
- **Logic**:
  - Validates the user's passcode using _verifyPasscode.
  - Checks OTP and biometric verification statuses.
  - Transfers Ether to the user and resets OTP and biometric statuses.
- **Event Emitted**: FundsWithdrawn(address user, uint256 amount).

6. Fallback

receive() external payable { ... }

- **Description**: Allows the wallet to accept Ether deposits directly.
- **Event Emitted**: FundsDeposited(address user, uint256 amount).

Internal Functions

_verifyPasscode

function _verifyPasscode(address user, string memory passcode) private view returns (bool) { ... }

- **Description**: Compares the user's provided passcode (hashed off-chain) with the stored on-chain hash.
- **Logic**: Uses keccak256 for hash comparison.
- **Returns**: true if the passcodes match, otherwise false.

Events

- FundsDeposited(address indexed user, uint256 amount): Logs deposits.
- FundsWithdrawn(address indexed user, uint256 amount): Logs withdrawals.
- PasscodeSet(address indexed user, bytes32 passcodeHash): Logs passcode updates.
- OTPVerified(address indexed user): Logs OTP verification.
- BiometricVerified(address indexed user): Logs biometric verification.

5. Deployment Guide

Dependencies

- No external dependencies are required for on-chain functionality.
- Off-chain services must be integrated for:
    - OTP Verification.
    - Biometric Authentication.

Deployment Steps

1. Deploy the SmartWallet contract.
2. Set up off-chain services for OTP and biometric verification.
3. Integrate the contract's verifyOTP and verifyBiometric functions with the off-chain services.

6. Testing Scenarios

1. Deposit

- **Input**: User deposits 1 ETH.
- **Expected Outcome**:
    - Contract balance increases by 1 ETH.
    - FundsDeposited event is emitted.

2. Set Passcode

- **Input**: User sets a passcode hash.
- **Expected Outcome**:
    - Passcode hash is stored for the user.
    - PasscodeSet event is emitted.

3. Withdraw with All Verifications

- **Input**: User withdraws 0.5 ETH after setting a passcode and verifying OTP and biometric.
- **Expected Outcome**:
    - User receives 0.5 ETH.
    - OTP and biometric verification statuses are reset.
    - FundsWithdrawn event is emitted.

4. Withdraw Without All Verifications

- **Input**: User withdraws without OTP or biometric verification.
- **Expected Outcome**:
  - Transaction is reverted.
  - Error message: OTP not verified or Biometric not verified.

7. Security Considerations

- **Access Control**:
  - OTP and biometric verification must be conducted through secure off-chain services.
  - Passcodes are hashed on-chain to ensure privacy.
- **Reentrancy Protection**:
  - Since the contract only deals with Ether withdrawals, reentrancy attacks are mitigated by the absence of external calls.
- **Input Validation**:
  - Requires non-zero deposit and withdrawal amounts.
  - Ensures only verified users can withdraw funds.

8. Future Improvements

- **Gas Optimization**:
  - Combine authentication checks into a single function to reduce gas costs.
- **Interoperability**:
  - Add support for ERC20 token deposits and withdrawals.
- **Recovery Mechanism**:
  - Implement a recovery mechanism in case of passcode or verification failures.