Writing the Contract

Let's get started with writing our first FHE powered contract.

Let's create a new file incontracts/ , and call thatWrappingERC20.sol .

touch contracts/WrappingERC20.sol Our goal is to create an ERC20 contract that supports shielded balances. Let's run through the different functions, step-by-step and show how we can implement each. We'll also link to more detailed explanations about custom functionality we make use of.

Importing FHE Librariesâ

// SPDX-License-Identifier: MIT pragma solidity ^ 0.8 .20 ;

import

"@openzeppelin/contracts/token/ERC20/ERC20.sol" ; import

"@fhenixprotocol/contracts/FHE.sol" ; The OpenZeppelin ERC20 contract will provide the basic functionality of the ERC20 token, whileFHE.sol is necessary to create and use FHE.

We'll also have to install the OpenZeppelin contracts, since they are not part of the default template.

- npm
- yarn
- pnpm

npm install @openzeppelin/contracts yarn install @openzeppelin/contracts pnpm install @openzeppelin/contracts

Creating the Contractâ

Inherit from ERC20â

The contract WrappingERC20 is an ERC20 contract. It uses function calls fromFHE.sol for encryption and to keep balances private and only viewable by the holder of the correct decryption key.

contract WrappingERC20 is ERC20

{

}

Create Encrypted Balancesâ

An encrypted balance is initialized for each address,_encBalances , which will hold encrypted token balances for users. TheeuintXXX are encrypted data types that represent FHE-encrypted unsigned integers of various bit lengths.

mapping (address

=> euint32) internal _encBalances ;

Constructorâ

The constructor function sets the name and symbol of the token, and then mints an initial 100 tokens to the address that deploys the contract.

constructor (string memory name , string memory symbol)

ERC20 (name , symbol)

{ _mint (msg . sender ,

100

*

10

**

uint (decimals ( ))) ; }

Wrapâ

First, let's define a functionwrap(uint32 amount) that allows users to convert (wrap) their tokens into encrypted form. The function will burn a specified amount from the user's balance and add the same amount to their encrypted balance.

function

wrap (uint32 amount)

public

{ // Make sure that the sender has enough of the public balance require (balanceOf (msg . sender)

    = amount) ; // Burn public balance _burn (msg . sender , amount) ;

// convert public amount to shielded by encrypting it euint32 shieldedAmount =

FHE . asEuint32 (amount) ; // Add shielded balance to his current balance _encBalances [msg . sender]

= _encBalances [msg . sender]

+ shieldedAmount ; } Breaking this down, the following logic is performed:

1. Verify that the user has enough public tokens to wrap
2. Burn public tokens
3. Add shielded tokens to the caller's balance

There are two main FHE operations that happened here:

- FHE.asEuint32(amount)
-
    - this converted a standard, publicuint
- to an FHE-encrypted number
- _encBalances[msg.sender] + shieldedAmount
-
    - this performs homomorphic addition between the two encrypted numbersshieldedAmount
- and_encBalances[msg.sender]

note Even though we calledFHE.asEuint32() on a public value and encrypted it we did not actually hide any information - the plaintext value was already known beforehand

Unwrapâ

Next, let's defineunwrap(inEuint32 amount) . This function will allow users to convert (unwrap) their encrypted tokens back into public tokens. The function will remove the specified amount from the user's encrypted balance and add the same amount to the user's public balance.

function

unwrap (inEuint32 memory amount)

public

{ euint32 _amount =

FHE . asEuint32 (amount) ; // verify that our shielded balance is greater or equal than the requested amount. (gte = greater-than-or-equal) FHE . req (_encBalances [msg . sender] . gte (_amount)) ; // subtract amount from shielded balance _encBalances [msg . sender]

= _encBalances [msg . sender]

- _amount ; // add amount to caller's public balance by calling themint function _mint (msg . sender ,

FHE . decrypt (_amount)) ; } Here we can see a few interesting things:

- FHE.req
- (stands for FHE require) verifies that a statement is true, or reverts the function. We use this to verify that we have enough shielded amount.
- _encBalances[msg.sender].gte(_amount)

- checks that_encBalances[msg.sender]
- isg
- reat
- er ore
- qual than_amount
- inEuint32
- is a data type specifically for input parameters. You can read more about ithere
- .

Encrypted Transfersâ

transferEncrypted(address to, bytes calldata encryptedAmount) is a public function that transfers encrypted tokens from the function caller to the to address. It converts the encrypted amount into the encrypted integer formeuint32 using theFHE.asEuint32(encryptedAmount) function and then calls_transferEncrypted . The function_transferEncrypted(address to, euint32 amount) is an internal function that just calls _transferImpl._transferImpl(address from, address to, euint32 amount) performs the actual transfer. It checks if the sender has enough tokens, then adds the amount to the to address encrypted balance and subtracts the same amount from the from address encrypted balance.

function

transferEncrypted (address to , inEuint32 calldata encryptedAmount)

public

{ euint32 amount =

FHE . asEuint32 (encryptedAmount) ; // Make sure the sender has enough tokens. (lte = less-then-or-equal) FHE . req (amount . lte (_encBalances [msg . sender])) ;

// Add to the balance ofto and subract from the balance ofmsg.sender. _encBalances [to]

= _encBalances [to]

+ amount ; _encBalances [msg . sender]

= _encBalances [msg . sender]

- amount ; } And that's it! To recap, we just created a contract that allows users to wrap and unwrap their tokens, and transfer them in encrypted form.

Let's see what the entire code looks like:

// SPDX-License-Identifier: MIT pragma solidity ^ 0.8 .20 ;

import

"@openzeppelin/contracts/token/ERC20/ERC20.sol" ; import

"@fhenixprotocol/contracts/FHE.sol" ;

contract WrappingERC20 is ERC20

{

mapping (address

=> euint32) internal _encBalances ;

constructor (string memory name , string memory symbol)

ERC20 (name , symbol)

{ _mint (msg . sender ,

100

*

10

**

uint (decimals ( ))) ; }

function

wrap (uint32 amount)

public

{ // Make sure that the sender has enough of the public balance require (balanceOf (msg . sender)

    = amount) ; // Burn public balance _burn (msg . sender , amount) ;

// convert public amount to shielded by encrypting it euint32 shieldedAmount =

FHE . asEuint32 (amount) ; // Add shielded balance to his current balance _encBalances [msg . sender]

= _encBalances [msg . sender]

+ shieldedAmount ; }

function

unwrap (inEuint32 memory amount)

public

{ euint32 _amount =

FHE . asEuint32 (amount) ; // verify that our shielded balance is greater or equal than the requested amount FHE . req (_encBalances [msg . sender] . gte (_amount)) ; // subtract amount from shielded balance _encBalances [msg . sender]

= _encBalances [msg . sender]

- _amount ; // add amount to caller's public balance by calling themint function _mint (msg . sender ,

FHE . decrypt (_amount)) ; }

function

transferEncrypted (address to , inEuint32 calldata encryptedAmount)

public

{ euint32 amount =

FHE . asEuint32 (encryptedAmount) ; // Make sure the sender has enough tokens. FHE . req (amount . lte (_encBalances [msg . sender])) ;

// Add to the balance ofto and subract from the balance offrom. _encBalances [to]

= _encBalances [to]

+ amount ; _encBalances [msg . sender]

= _encBalances [msg . sender]

- amount ; } } Note that in a real use case the actual code would include more functionality, and structure things a bit differently. If you want to see what such a contract looks like, you can check out theFHERC20 contract in the Fhenix contracts repository.

Wait a second...â

But what about viewing the encrypted balances? Well, we'll cover that in the next section, where we'll be adding viewing functionality to our contract, and see how we can utilizePermissions to manage access to encrypted data. Edit this page