(Un)Sealing

When an app wants to read some piece of encrypted data from a Fhenix smart contract, that data must be converted from its encrypted form on chain to an encryption that the app or user can read.

The process of taking an FHE-encrypted ciphertext and converting it to standard encryption is calledsealing.

The data is returned to the user usingsealed box encryption from NaCL. The gist of it is that the user provides a public key to the contract during a view function call, which the contract then uses to encrypt the data in such a way that only the owner of the private key associated with the provided public key can decrypt and read the data.

Don't Want to Seal? Fhenix supports standard decryption as well. Mostly suited for public data, an unsealed plaintext value can be returned from a contract. You can read more about how to do thishere .

Encrypted Values & Permits

When reading encrypted values we can do one of two things:

- Receiving it as bytes calldata: 0x04000....
- RECOMENDED: Receiving it as inEuint*: ["0x04000"]

The main difference with inEuint* is that you can be explicit with what is the exact parameter that you are looking for.

APermit is a data structure that helps contracts know who is trying to call a specific function.

The fhenix.js Javascript library includes methods to support creating parameters for values that requirePermits & Access Control. These methods can help creating ephemeral transaction keys, which are used by the smart contract to create a secure encryption channel to the caller. Similarly to decryption, this usage can be implemented by any compliant library, but we include direct support in fhenix.js.

This is done in 3 steps: generating a permit, querying the contract and unsealing the data.

1. Creating a Permit

import

{

FhenixClient , getPermit }

from

'fhenixjs' ;

const provider =

new

ethers . JsonRpcProvider ('https://test01.fhenix.zone/evm') ; const client =

new

FhenixClient ({ provider }) ;

const permit =

await

getPermit (contractAddress , provider) ; client . storePermit (permit) ; Did you know? When you create a permit it gets stored inlocalstorage . This makes permits easily reusable and transferable

2. Querying the Contract

We recommend that contracts implement the Permit/Permission interfaces (though this is not strictly required!). In this case, we can easily inject our permit into the function call.

const permission = client . extractPermitPermission (permit) ; const response =

await contract . balanceOf (permission) ;

3. Unsealing the Data

Now that we have the response data, we can use theunseal function to decipher the data

client . unseal (contractAddress , response) We have to provide the contract address so the fhenix client knows which permit to use for the unsealing function.

note Permits are currently limited to support a single contract

Putting it all Together

import

{ FhenixClient , getPermit }

from

'fhenixjs' ; import

{ JsonRpcProvider }

from

'ethers' ;

const provider =

new

ethers . JsonRpcProvider ('https://test01.fhenix.zone/evm') ; const client =

new

FhenixClient ({ provider }) ;

const permit =

await

getPermit (contractAddress , provider) ; client . storePermit (permit) ;

const permission = client . extractPermitPermission (permit) ; const response =

await contract . balanceOf (permission) ;

const plaintext = client . unseal (contractAddress , response) ;

console . log (My Balance: { plaintext }) Did you know? You have tools that can ease the process of interacting with the contract and decrypting values. If you want to use them please refer toTools and Utilities Edit this page

Previous Encryption Next Permits