Permits & Access Control

In a Fully Homomorphic Encryption (FHE) context, data stored in the contract's storage is encrypted. Therefore, granting selective access to data becomes an essential part of access control. This is done via thesealoutput function, which seals the data in a manner that only the intended recipient can decrypt and view it (or thedecrypt function, for less sensitive data). This approach ensures that encrypted data remains confidential and only accessible to authorized users.

Usually, Solidity contracts will expose their data using view functions. However, in the context of permissioned data this is challenging since view functions do not come with any kind of mechanism to allow the contract to cryptographically verify that the caller is who he says he is - in the case of transactions this is done by verifying the signature on the data.

This means that we have to create mechanisms that enable the contract to determine who can access the data and who can't.

Permits

Permits are a mechanism that allows the contract to cryptographically verify that the caller is who he says he is.

Simply, they are a signed message that contains the caller's public key, which the contract can then use to verify that the caller is who he says he is.

Permits & Access Control (EIP-712)

Out-of-the-box, Fhenix Solidity libraries come with a basic access control scheme. This helps contracts perform a basic check for ownership of an account.

This makes it easy for contracts to add authentication & authorization to specific view functions, without having to reinvent the wheel every time.

This page will cover how access permits and permissions that are created and used in fhenix.js.

What is a Permit?

In the context of Fhenix and blockchain, a permit refers to a signed JSON object that follows the EIP-712 standard. This permit contains the necessary information, including a public key, that allows data re-sealing in a smart contract environment. The inclusion of this public key into the permit enables a secure process of data re-sealing within a smart contract once the JSON object is signed by the user.

How to Generate a Permit

Permits are generated using thegetPermit method. This method requires the following parameter:

- contractAddress
- (required, string): The address of the contract.
- provider
- (required): Note that if you want to unseal data using your wallet's encryption key you can't use "JsonRpcProvider" you will need to use a provider that can sign.

const permit =

await

getPermit (contractAddress) ;

What is a Permission?

In the context of Fhenix, a permission is the part of a permit that supplies the proof that the caller is who he says he is. A permission contains the signature and the corresponding public key. In order to see how to verify a permission in your solidity contract please refer to ourPermissioned .

How to Generate a Permission

const permission = client . extractPermitPermissions (permit) ;

Using a Permission

Once generated, you can use the permission and send it to the contract. You can also unseal the outputs of "sealoutput" assuming it was sealed using your permission.

```
import

{

BrowserProvider

}

from

"ethers" ; import

{

FhenixClient , getPermit }

from

"fhenixjs" ;

const provider =

new

BrowserProvider ( window . ethereum ) ; const client =

new

FhenixClient ( { provider } ) ; const permit =

await

getPermit ( contractAddress , provider ) ; const permission = client . extractPemitPermissions ( permit ) ; client . storePermit ( permit ) ;

// Stores a permit for a specific contract address. const response =

await contract . connect ( owner ) . getValue ( permission ) ;

// Calling "getValue" which is a view function in "contract" const plaintext =

await client . unseal ( contractAddress , response ) ;
```

Edit this page

Previous Examples & fheDapps Next Privacy Pitfalls