Permits & Permissions

Overview

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.

Usage

Permits are meant to be used together with the interfaces exposed byPermissioned.Sol . If a contract expects aSignature parameter, that's a good sign that we should use apermit to manage and create user permissions.

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.

To confirm whether the recipient is authorized, EIP712 signatures are employed. EIP712 is a standard for Ethereum signed messages that makes it easier to understand the information being signed. This allows us to verify that the signer of a given piece of data is the owner of the account they claim to be.

Did You Know? When signing EIP712 typed data, wallets such as MetaMask provide a more transparent, safe interface for users to understand what they are signing Let's see this concept in action using an example. In an encrypted ERC20 token contract, a user would want to query their token balance. Since the balance is stored as encrypted data, the contract must first verify that the query is indeed from the token owner before revealing the information. This is where the EIP712 signatures step in.

Below is a function from an EncryptedERC20 contract:

function

balanceOf (Permission calldata perm) public view onlySender (perm) returns

(bytes memory) { return

FHE . sealoutput (balances [msg . sender] , perm . publicKey) ; } In this function,onlySender is a modifier that verifies if the EIP712 signature is valid. If the signature corresponds to the account that is making the call (msg.sender), then the function will execute. If not, it will revert.

Here's what theonlySender modifier looks like:

struct Permission

{ bytes32 publicKey ; bytes signature ; }

modifier onlySender (Permission memory permission)

{ bytes32 digest =

_hashTypedDataV4 (keccak256 (abi . encode (keccak256 ("Permissioned(bytes32 publicKey)") , permission . publicKey))) ; address signer =

ECDSA . recover (digest , permission . signature) ; if

(signer != msg . sender) revert SignerNotMessageSender ( ) ; _ ; } TheonlySender modifier takes aPermission . It then calculates thedigest from thepublicKey . The signer's address is recovered from thedigest using theECDSA.recover function. If the recovered address matchesmsg.sender , it means that the caller is indeed the owner of the account and is allowed to access the data.

You can use this helpful contract out-of-the-box by importing it from@fhenixprotocol/contracts/access and can be easily imported to integrate into your contracts.

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

import

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

"@fhenixprotocol/contracts/Fhe.sol" ; import

{

Permissioned

}

from

"@fhenixprotocol/contracts/access/Permissioned.sol" ;

contract WrappingERC20 is Permissioned ,

ERC20

{

function

balanceOfEncrypted (Permission memory perm) public view onlySender (perm) returns

(bytes memory)

{ return

FHE . sealoutput (_encBalances [msg . sender] , perm . publicKey) ; } } For a full example what this looks like - seeEncryptedERC20.sol or ourgetting started tutorial for a full example, including client-side integration.

Advanced Access Control

While the above-mentioned access control scheme leveraging EIP712 signatures provides a robust mechanism for verifying the identity of users querying encrypted data, it does have some limitations. One of the primary missing pieces is the absence of roles and permissions associated with those roles. The scheme as described validates that a user querying a balance, for example, is indeed the owner of that account, but it doesn't provide a mechanism for defining different levels of access or permissions.

For instance, in more complex scenarios, you might want to allow certain users to only view specific pieces of data, or perhaps perform certain actions based on their role (admin, user, auditor, etc.). Moreover, there's no provision for dynamic access control in which permissions could be granted or revoked at runtime.

Additionally, this scheme doesn't cover collective authority, where, for example, an action might require the approval of multiple participants to be executed. Such advanced access control mechanisms are not built into this scheme and would need to be implemented separately based on specific application needs.

Lastly, the EIP-712 standard mostly considers messages targeted at a single smart contract. Some use-cases, however, benefit from allowing the user to provide access to multiple contracts concurrently. For example, consider a DeX (decentralized exchange). Allowing such an app to be able to display the balances of all the user's different tokens would be a UX challenge if the user had to approve each one individually.

Standardization

While we recommend and provide Permits as a basic access control mechanism, we do not enforce any particular standard for them. We feel that as the ecosystem evolves, different standards will emerge and we do not want to limit the ecosystem by enforcing a particular standard at this stage.

In other words, if you think there is a better way to do it, feel free to do soEdit this page

Previous (Un)Sealing Next End-to-End Example