Depositing Tokens to Aztec

In this step, we will write our token portal contract on L1.

Initialize Solidity contract

Inl1-contracts/contracts in your file calledTokenPortal.sol paste this:

pragma

solidity

= 0.8.18 ;

import

{ IERC20 }

from

"@openzeppelin/contracts/token/ERC20/IERC20.sol" ; import

{ SafeERC20 }

from

"@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol" ;

// Messaging import

{ IRegistry }

from

"@aztec/l1-contracts/src/core/interfaces/messagebridge/IRegistry.sol" ; import

{ IInbox }

from

"@aztec/l1-contracts/src/core/interfaces/messagebridge/IInbox.sol" ; import

{ DataStructures }

from

"@aztec/l1-contracts/src/core/libraries/DataStructures.sol" ; import

{ Hash }

from

"@aztec/l1-contracts/src/core/libraries/Hash.sol" ;

contract

TokenPortal

{ using

SafeERC20

for IERC20 ;

IRegistry public registry ; IERC20 public underlying ; bytes32

public l2TokenAddress ;

function

initialize (address _registry ,

address _underlying ,

bytes32 _l2TokenAddress)

external

{ registry =

IRegistry (_registry) ; underlying =

IERC20 (_underlying) ; l2TokenAddress = _l2TokenAddress ; } This imports relevant files including the interfaces used by the Aztec rollup. And initializes the contract with the following parameters:

- rollup registry address (that stores the current rollup, inbox and outbox contract addresses)
- The ERC20 token the portal corresponds to
- The address of the sister contract on Aztec to where the token will send messages to (for depositing tokens or from where to withdraw the tokens)

Create a basic ERC20 contract that can mint tokens to anyone. We will use this to test.

Create a filePortalERC20.sol in the same folder and add:

contract pragma

solidity

^ 0.8.0 ;

import

"@oz/token/ERC20/ERC20.sol" ;

contract

PortalERC20

is ERC20 { constructor ( )

ERC20 ("Portal" ,

"PORTAL")

{ }

function

mint (address to ,

uint256 amount)

external

{ _mint (to , amount) ; } }Source code: l1-contracts/test/portals/PortalERC20.sol#L2-L14

Depositing tokens to Aztec publicly

Next, we will write a function that is used to deposit funds on L1 that a user may have into an Aztec portal and send a message to the Aztec rollup to mint tokenspublicly on Aztec.

Paste this inTokenPortal.sol

deposit_public /* ** @notice Deposit funds into the portal and adds an L2 message which can only be consumed publicly on Aztec * @param _to - The aztec address of the recipient * @param _amount - The amount to deposit * @param _canceller - The address that can cancel the L1 to L2 message * @param _deadline - The timestamp after which the entry can be cancelled * @param _secretHash - The hash of the secret consumable message. The hash should be 254 bits (so it can fit in a Field element) * @return The key of the entry in the Inbox* / function

depositToAztecPublic (bytes32 _to , uint256 _amount , address _canceller , uint32 _deadline , bytes32 _secretHash)

external

payable

returns

(bytes32)

{ // Preamble IInbox inbox = registry . getInbox ( ) ; DataStructures . L2Actor memory actor = DataStructures . L2Actor (l2TokenAddress ,

1) ;

// Hash the message content to be reconstructed in the receiving contract bytes32 contentHash = Hash . sha256ToField (abi . encodeWithSignature ("mint_public(bytes32,uint256,address)" , _to , _amount , _canceller)) ;

// Hold the tokens in the portal underlying . safeTransferFrom (msg . sender ,

address (this) , _amount) ;

// Send message to rollup return inbox . sendL2Message { value : msg . value } (actor , _deadline , contentHash , _secretHash) ; } Source code: l1-contracts/test/portals/TokenPortal.sol#L29-L61 Here is an explanation of what it is doing:

1. We first ask the registry for the inbox contract address (to which we send messages to)
2. We construct the “content” of the message we need to send to the recipient on Aztec.*The content is limited to a single field (~254 bits). So if the content is larger, we have to hash it and the hash can be passed along.* We use our utility method that creates a sha256 hash but truncates it to fit into a field
3.
   - Since we want to mint tokens on Aztec publicly, the content here is the amount to mint and the address on Aztec who will receive the tokens. We also include the L1 address that can cancel the L1->L2 message. Adding this into the content hash makes it so that only the appropriate person can cancel the message and not just any malicious 3rd party.* More on cancellers can be found inthis upcoming section
4.
   - We encode this message as a mint_public function call, to specify the exact intentions and parameters we want to execute on L2.* In reality the content can be constructed in any manner as long as the sister contract on L2 can also create it. But for clarity, we are constructing the content like a abi encoded function call.
5.
   -
     - It is good practice to include all parameters used by L2 into this content (like the amount and to) so that a malicious actor can’t change the to to themselves when consuming the message.
6. The tokens are transferred from the user to the portal usingunderlying.safeTransferFrom()
7. . This puts the funds under the portal's control.
8. Next we send the message to the inbox contract. The inbox expects the following parameters:* recipient (calledactor
9.
   - here), a struct:* the sister contract address on L2 that can consume the message.
10.
    -
      - The version - akin to THE chainID of Ethereum. By including a version, an ID, we can prevent replay attacks of the message (without this the same message might be replayable on other aztec networks that might exist).
11.
    - Deadline by which the sequencer on L2 must consume the method. After this time, the message can be canceled by the “canceller”. We will implement this functionality later in the doc.
12.
    - A secret hash (fit to a field element). This is mainly used in the private domain and the preimage of the hash doesn’t need to be secret for the public flow. When consuming the message, one must provide the preimage. More on this when we create the private flow for depositing tokens.
13.
    - We also pass a fee to the sequencer for including the message. It is a uint64.
14. It returns abytes32 key
15. which is the id for this message in the Inbox.

So in summary, it deposits tokens to the portal, encodes a mint message, hashes it, and sends it to the Aztec rollup via the Inbox. The L2 token contract can then mint the tokens when it processes the message.

Depositing tokens to Aztec privately

Let’s do the similar for the private flow:

deposit_private /* *@notice Deposit funds into the portal and adds an L2 message which can only be consumed privately on Aztec * @param _secretHashForRedeemingMintedNotes - The hash of the secret to redeem minted notes privately on Aztec. The hash should be 254 bits (so it can fit in a Field element) * @param _amount - The amount to deposit * @param _canceller - The address that can cancel the L1 to L2 message * @param _deadline - The timestamp after which the entry* *can be cancelled * @param _secretHashForL2MessageConsumption - The hash of the secret consumable L1 to L2 message. The hash should be 254 bits (so it can fit in a Field element) * @return The key of the entry in the Inbox* / function

depositToAztecPrivate (bytes32 _secretHashForRedeemingMintedNotes , uint256 _amount , address _canceller , uint32 _deadline , bytes32 _secretHashForL2MessageConsumption)

external

payable

returns

(bytes32)

{ // Preamble IInbox inbox = registry . getInbox ( ) ; DataStructures . L2Actor memory actor = DataStructures . L2Actor (l2TokenAddress ,

1) ;

// Hash the message content to be reconstructed in the receiving contract bytes32 contentHash = Hash . sha256ToField (abi . encodeWithSignature ("mint_private(bytes32,uint256,address)" , _secretHashForRedeemingMintedNotes , _amount , _canceller)) ;

// Hold the tokens in the portal underlying . safeTransferFrom (msg . sender ,

address (this) , _amount) ;

// Send message to rollup return inbox . sendL2Message { value : msg . value } (actor , _deadline , contentHash , _secretHashForL2MessageConsumption) ; } Source code: l1-contracts/test/portals/TokenPortal.sol#L63-L102 Here we want to send a message to mint tokens privately on Aztec! Some key differences from the previous method are:

- The content hash uses a different function name -mint_private
- . This is done to make it easy to separate concerns. If the contentHash between the public and private message was the same, then an attacker could consume a private message publicly!
- Since we want to mint tokens privately, we shouldn't specify ato
- Aztec address (remember that Ethereum is completely public). Instead, we will use a secret hash -secretHashForRedeemingMintedNotes
- . Only he who knows the preimage to the secret hash can actually mint the notes. This is similar to the mechanism we use for message consumption on L2
- Like with the public flow, we move the user's funds to the portal
- We now send the message to the inbox with thefee
- ,deadline
- , therecipient
- (the sister contract on L2 along with the version of aztec the message is intended for) and thesecretHashForL2MessageConsumption
- (such that on L2, the consumption of the message can be private).

Note that because L1 is public, everyone can inspect and figure out the fee, contentHash, deadline, recipient contract address.

So how do we privately consume the message on Aztec?

On Aztec, anytime something is consumed (i.e. deleted), we emit a nullifier hash and add it to the nullifier tree. This prevents double-spends. The nullifier hash is a hash of the message that is consumed. So without the secret, one could reverse engineer the expected nullifier hash that might be emitted on L2 upon message consumption. To consume the message on L2, the user provides a secret to the private function, which computes the hash and asserts that it matches to what was provided in the L1->L2 message. This secret is included in the nullifier hash computation and the nullifier is added to the nullifier tree. Anyone inspecting the blockchain won't know which nullifier hash corresponds to the L1->L2 message consumption.

note Secret hashes are Pedersen hashes since the hash has to be computed on L2 and sha256 hash is very expensive for zk circuits. The content hash however is a sha256 hash truncated to a field as shown before. In the next step we will start writing our L2 smart contract to mint these tokens on L2. Edit this page

Previous Setup and Installation Next Minting tokens on Aztec