Withdrawing to L1

This is where we have tokens on Aztec and want to withdraw them back to L1 (i.e. burn them on L2 and mint on L1). Withdrawing from L1 will be public.

Withdrawing publicly

Go back to yourmain.nr and paste this:

exit_to_l1_public // Burns the appropriate amount of tokens and creates a L2 to L1 withdraw message publicly // Requires msg.sender to give approval to the bridge to burn tokens on their behalf using witness signatures

[aztec(public)]

fn

exit_to_l1_public (recipient :

EthAddress ,

// ethereum address to withdraw to amount :

Field , callerOnL1 :

EthAddress ,

// ethereum address that can call this function on the L1 portal (0x0 if anyone can call) nonce :

Field

// nonce used in the approval message by msg.sender to let bridge burn their tokens on L2)

{ // Send an L2 to L1 message let content =

get_withdraw_content_hash (recipient , amount , callerOnL1) ; context . message_portal (context . this_portal_address ( ) , content) ;

// Burn tokens Token :: at (storage . token . read ( )) . burn_public (& mut context , context . msg_sender ( ) , amount , nonce) ; } Source code: noir-projects/noir-contracts/contracts/token_bridge_contract/src/main.nr#L48-L65 For this to work we import theget_withdraw_content_hash helper function from thetoken_portal_content_hash_lib .

What's happening here?

Theexit_to_l1_public function enables anyone to withdraw their L2 tokens back to L1 publicly. This is done by burning tokens on L2 and then creating an L2->L1 message.

1. Like with our deposit function, we need to create the L2 to L1 message. The content is theamount
2. to burn, the recipient address, and who can execute the withdraw on the L1 portal on behalf of the user. It can be0x0
3. for anyone, or a specified address.
4. context.message_portal()
5. passes this content to thekernel circuit
6. which creates the proof for the transaction. The kernel circuit then adds the sender (the L2 address of the bridge + version of aztec) and the recipient (the portal to the L2 address + the chain ID of L1) under the hood, to create the message which gets added as part of the transaction data published by the sequencer and is stored in the outbox for consumption.
7. Finally, you also burn the tokens on L2! Note that it burning is done at the end to follow the check effects interaction pattern. Note that the caller has to first approve the bridge contract to burn tokens on its behalf. Refer toburn_public function on the token contract
8. . The nonce parameter refers to the approval message that the user creates - also refer toauthorizing token spends here
9. .* We burn the tokens from themsg_sender()
10.
    - . Otherwise, a malicious user could burn someone else's tokens and mint tokens on L1 to themselves. One could add another approval flow on the bridge but that might make it complex for other applications to call the bridge.

Withdrawing Privately

This function works very similarly to the public version, except here we burn user's private notes. Under the public function in yourmain.nr , paste this:

exit_to_l1_private // Burns the appropriate amount of tokens and creates a L2 to L1 withdraw message privately // Requires msg.sender (caller of the method) to give approval to the bridge to burn tokens on their behalf using witness signatures

[aztec(private)]

fn

exit_to_l1_private (token :

AztecAddress , recipient :

EthAddress ,

// ethereum address to withdraw to amount :

Field , callerOnL1 :

EthAddress ,

// ethereum address that can call this function on the L1 portal (0x0 if anyone can call) nonce :

Field

// nonce used in the approval message by msg.sender to let bridge burn their tokens on L2)

{ // Send an L2 to L1 message let content =

get_withdraw_content_hash (recipient , amount , callerOnL1) ; context . message_portal (context . this_portal_address ( ) , content) ;

// Assert that user provided token address is same as seen in storage. context . call_public_function (context . this_address ( ) , FunctionSelector :: from_signature ("_assert_token_is_same((Field))") , [token . to_field ( )]) ;

// Burn tokens Token :: at (token) . burn (& mut context , context . msg_sender ( ) , amount , nonce) ;Source code: noir-projects/noir-contracts/contracts/token_bridge_contract/src/main.nr#L96-L123 assert_token_is_same

[aztec(public)]

[aztec(internal)]

fn

_assert_token_is_same (token :

AztecAddress)

{ assert (storage . token . read ( ) . eq (token) ,

"Token address is not the same as seen in storage") ; }Source code: noir-projects/noir-contracts/contracts/token_bridge_contract/src/main.nr#L158-L164 Since this is a private method, it can't read what token is publicly stored. So instead the user passes a token address, and_assert_token_is_same() checks that this user provided address is same as the one in storage.

Because public functions are executed by the sequencer while private methods are executed locally, all public calls are always doneafter all private calls are completed. So first the burn would happen and only later the sequencer asserts that the token is same. The sequencer just sees a request toexecute_assert_token_is_same and therefore has no context on what the appropriate private method was. If the assertion fails, then the kernel circuit will fail to create a proof and hence the transaction will be dropped.

Once again, a user must sign an approval message to let the contract burn tokens on their behalf. The nonce refers to this approval message.

For both the public and private flow, we use the same mechanism to determine the content hash. This is because on L1, things are public anyway. The only different between the two functions is that in the private domain we have to nullify user's notes where as in the public domain we subtract the balance from the user.

Withdrawing on L1

After the transaction is completed on L2, the portal must call the outbox to successfully transfer funds to the user on L1. Like with deposits, things can be complex here. For example, what happens if the transaction was done on L2 to burn tokens but can't be withdrawn to L1? Then the funds are lost forever! How do we prevent this?

Paste this in yourTokenPortal.sol :

/* * @notice Withdraw funds from the portal * @dev Second part of withdraw, must be initiated from L2 first as it will consume a message from outbox * @param _recipient - The address to send the funds to * @param _amount - The amount to withdraw * @param _withCaller - Flag to use msg.sender as caller, otherwise address(0) * Must match the caller of the message (specified from L2) to consume it. * @return The key of the entry in the Outbox / function

withdraw (address _recipient ,

uint256 _amount ,

bool _withCaller) external returns

(bytes32) { DataStructures . L2ToL1Msg memory message = DataStructures . L2ToL1Msg ({ sender : DataStructures . L2Actor (l2TokenAddress ,

1) , recipient : DataStructures . L1Actor (address (this) , block . chainid) , content : Hash . sha256ToField (abi . encodeWithSignature ("withdraw(address,uint256,address)" , _recipient , _amount , _withCaller ? msg . sender :

address (0))) }) ;

bytes32 entryKey = registry . getOutbox ( ) . consume (message) ;

underlying . transfer (_recipient , _amount) ;

return entryKey ; } } Here we reconstruct the L2 to L1 message and check that this message exists on the outbox. If so, we consume it and transfer the funds to the recipient. As part of the reconstruction, the content hash looks similar to what we did in our bridge contract on aztec where we pass the amount and recipient to the the hash. This way a malicious actor can't change the recipient parameter to the address and withdraw funds to themselves.

We also use a_withCaller parameter to determine the appropriate party that can execute this function on behalf of the recipient. IfwithCaller is false, then anyone can call the method and hence we use address(0), otherwise only msg.sender should be able to execute. This address should match thecallerOnL1 address we passed in aztec when withdrawing from L2.

We call this patterndesigned caller which enables a new paradigmwhere we can construct other such portals that talk to the token portal and therefore create more seamless crosschain legos between L1 and L2.

Before we can compile and use the contract, we need to add two additional functions.

We need a function that lets us read the token value. Paste this intomain.nr :

read_token unconstrained fn

token ( )

->

pub

AztecAddress

{ storage . token . read ( ) }Source code: noir-projects/noir-contracts/contracts/token_bridge_contract/src/main.nr#L134-L138

Compile code

Congratulations, you have written all the contracts we need for this tutorial! Now let's compile them.

Compile your Solidity contracts using hardhat. Run this in thepackages directory:

cd l1-contracts npx hardhat compile And compile your Aztec.nr contracts like this:

cd aztec-contracts/token_bridge aztec-nargo compile And generate the TypeScript interface for the contract and add it to the test dir:

aztec-cli codegen target -o .. / .. /src/test/fixtures --ts This will create a TS interface in oursrc/test folder!

In the next step we will write the TypeScript code to deploy our contracts and call on both L1 and L2 so we can see how everything works together. Edit this page

Previous Cancelling Deposits Next Deploy & Call Contracts with Typescript