# L2 Contract Setup

As we mentioned inthe overview , the Uniswap L2 contract will receive funds from the user and then burn funds on L2 to withdraw. To do this it callsTokenBridge.exit_to_l1_public() which burns funds on the Uniswap contract. The bridge needs approval from the Uniswap contract to burn its funds.

In this step, we will set up the storage struct for our Uniswap contract and define the functions to approve and validate burn actions.

Our main contract will live insideuniswap/src/main.nr . Inmain.nr , paste this initial setup code:

uniswap_setup mod

interfaces ; mod

util ;

// Demonstrates how to use portal contracts to swap on L1 Uniswap with funds on L2 // Has two separate flows for private and public respectively // Uses the token bridge contract, which tells which input token we need to talk to and handles the exit funds to L1 contract Uniswap

{ use

dep :: aztec :: prelude :: { FunctionSelector ,

AztecAddress ,

EthAddress ,

Map ,

PublicMutable } ; use

dep :: aztec :: oracle :: context :: get_portal_address ;

use

dep :: authwit :: auth :: { IS_VALID_SELECTOR , assert_current_call_valid_authwit_public , compute_call_authwit_hash , compute_outer_authwit_hash } ;

use

crate :: interfaces :: { Token ,

TokenBridge } ; use

crate :: util :: { compute_swap_private_content_hash , compute_swap_public_content_hash } ;

struct

Storage

{ // like with account contracts, stores the approval message on a slot and tracks if they are active approved_action :

Map < Field ,

PublicMutable < bool

, // tracks the nonce used to create the approval message for burning funds // gets incremented each time after use to prevent replay attacks nonce_for_burn_approval :

PublicMutable < Field

, } Source code: noir-projects/noir-contracts/contracts/uniswap_contract/src/main.nr#L1-L27 What's happening here?

Because Uniswap has to approve the bridge to withdraw funds, it has to handle the approvals. So it stores a map of all the actions that are approved. The approval message is hashed to a field and stored in the contract's storage in theapproved_action map.

To ensure there are no collisions (i.e. when the contract wants to approve the bridge of the exact same amount, the message hash would be the same), we also keep a nonce that gets incremented each time after use in a message.

## Building the approval flow

Next, paste this function:

authwit_uniswap_get // Since the token bridge burns funds on behalf of this contract, this contract has to tell the token contract if the signature is valid // implementation is similar to how account contracts validate public approvals. // if valid, it returns the IS_VALID selector which is expected by token contract

# [aztec(public)]

fn

spend_public_authwit ( inner_hash :

Field )

->

Field

{ let message_hash =

compute_outer_authwit_hash ( context . msg_sender ( ) , inner_hash ) ; let value = storage . approved_action . at ( message_hash ) . read ( ) ; if

( value )

{ context . push_new_nullifier ( message_hash ,

0 ) ; IS_VALID_SELECTOR }

else

{ 0 } }Source code: noir-projects/noir-contracts/contracts/uniswap_contract/src/main.nr#L171-L186 In this function, the token contract calls the Uniswap contract to check if Uniswap has indeed done the approval. The token contract expects aspend_private_authwit() function to exit for private approvals andspend_public_authwit() for public approvals. If the action is indeed approved, it expects that the contract will emit a nullifier and return the function selector forIS_VALID() in both cases. The Aztec.nr library exposes this constant for ease of use.

This is similar to theAuthwit flow .

However we don't have a function that actually creates the approved message and stores the action. This method should be responsible for creating the approval and then calling the token bridge to withdraw the funds to L1:

authwit_uniswap_set // This helper method approves the bridge to burn this contract's funds and exits the input asset to L1 // Assumes contract already has funds. // Assume token relates to token_bridge (ie token_bridge.token == token) // Note that private can't read public return values so created an internal public that handles everything // this method is used for both private and public swaps.

# [aztec(public)]

# [aztec(internal)]

fn

_approve_bridge_and_exit_input_asset_to_L1 ( token :

AztecAddress , token_bridge :

AztecAddress , amount :

Field )

{ // approve bridge to burn this contract's funds (required when exiting on L1, as it burns funds on L2): let nonce_for_burn_approval = storage . nonce_for_burn_approval . read ( ) ; let selector =

FunctionSelector :: from_signature ( "burn_public((Field),Field,Field)" ) ; let message_hash =

compute_call_authwit_hash ( token_bridge , token , selector , [ context . this_address ( ) . to_field ( ) , amount , nonce_for_burn_approval ] ) ; storage . approved_action . at ( message_hash ) . write ( true ) ;

// increment nonce_for_burn_approval so it won't be used again storage . nonce_for_burn_approval . write ( nonce_for_burn_approval +

1 ) ;

// Exit to L1 Uniswap Portal ! TokenBridge :: at ( token_bridge ) . exit_to_l1_public ( & mut context , context . this_portal_address ( ) , amount , context . this_portal_address ( ) , nonce_for_burn_approval ) ; } Source code: noir-projects/noir-contracts/contracts/uniswap_contract/src/main.nr#L188-L220 Notice how the nonce also gets incremented.

In the next step we'll go through a public swapping flow.Edit this page

Previous Uniswap Portal on L1 Next Swapping Publicly