# Swapping Privately

In theuniswap/src/main.nr contract we createdpreviously inaztec-contracts/uniswap , paste these functions:

swap_private

# [aztec(private)]

fn

swap_private ( input_asset :

AztecAddress ,

// since private, we pass here and later assert that this is as expected by input_bridge input_asset_bridge :

AztecAddress , input_amount :

Field , output_asset_bridge :

AztecAddress , // params for using the unshield approval nonce_for_unshield_approval :

Field , // params for the swap uniswap_fee_tier :

Field , // which uniswap tier to use (eg 3000 for 0.3% fee) minimum_output_amount :

Field ,

// minimum output amount to receive (slippage protection for the swap) // params for the depositing output_asset back to Aztec secret_hash_for_redeeming_minted_notes :

Field , // secret hash used to redeem minted notes at a later time. This enables anyone to call this function and mint tokens to a user on their behalf secret_hash_for_L1_to_l2_message :

Field ,

// for when l1 uniswap portal inserts the message to consume output assets on L2 deadline_for_L1_to_l2_message :

Field ,

// for when l1 uniswap portal inserts the message to consume output assets on L2 canceller_for_L1_to_L2_message :

EthAddress ,

// L1 address of who can cancel the message to consume assets on L2. caller_on_L1 :

EthAddress

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

{ // Assert that user provided token address is same as expected by token bridge. // we can't directly use input_asset_bridge.token because that is a public method and public can't return data to private context . call_public_function ( context . this_address ( ) , FunctionSelector :: from_signature ( "_assert_token_is_same((Field),(Field))" ) , [ input_asset . to_field ( ) , input_asset_bridge . to_field ( ) ] ) ;

// Transfer funds to this contract Token :: at ( input_asset ) . unshield ( & mut context , context . msg_sender ( ) , context . this_address ( ) , input_amount , nonce_for_unshield_approval ) ;

// Approve bridge to burn this contract's funds and exit to L1 Uniswap Portal context . call_public_function ( context . this_address ( ) , FunctionSelector :: from_signature ( "_approve_bridge_and_exit_input_asset_to_L1((Field),(Field),Field)" ) , [ input_asset . to_field ( ) , input_asset_bridge . to_field ( ) , input_amount ] ) ;

// Create swap message and send to Outbox for Uniswap Portal // this ensures the integrity of what the user originally intends to do on L1. let input_asset_bridge_portal_address =

get_portal_address ( input_asset_bridge ) ; let output_asset_bridge_portal_address =

get_portal_address ( output_asset_bridge ) ; // ensure portal exists - else funds might be lost assert ( ! input_asset_bridge_portal_address . is_zero ( ) , "L1 portal address of input_asset's bridge is 0" ) ; assert ( ! output_asset_bridge_portal_address . is_zero ( ) ,

"L1 portal address of output_asset's bridge is 0" ) ;

let content_hash =

compute_swap_private_content_hash ( input_asset_bridge_portal_address , input_amount , uniswap_fee_tier , output_asset_bridge_portal_address , minimum_output_amount , secret_hash_for_redeeming_minted_notes , secret_hash_for_L1_to_l2_message , deadline_for_L1_to_l2_message , canceller_for_L1_to_L2_message , caller_on_L1 ) ; context . message_portal ( context . this_portal_address ( ) , content_hash ) ; } Source code: noir-projects/noir-contracts/contracts/uniswap_contract/src/main.nr#L100-L169 assert_token_is_same

## [aztec(public)]

## [aztec(internal)]

fn

_assert_token_is_same ( token :

AztecAddress , token_bridge :

AztecAddress )

{ assert ( token . eq ( TokenBridge :: at ( token_bridge ) . token ( & mut context ) ) ,

"input_asset address is not the same as seen in the bridge contract" ) ; }Source code: noir-projects/noir-contracts/contracts/uniswap_contract/src/main.nr#L222-L230 This uses a util functioncompute_swap_private_content_hash() - let's add that.

Inutil.nr , add:

compute_swap_private_content_hash // This method computes the L2 to L1 message content hash for the private // refer l1-contracts/test/portals/UniswapPortal.sol on how L2 to L1 message is expected pub

fn

compute_swap_private_content_hash ( input_asset_bridge_portal_address :

EthAddress , input_amount :

Field , uniswap_fee_tier :

Field , output_asset_bridge_portal_address :

EthAddress , minimum_output_amount :

Field , secret_hash_for_redeeming_minted_notes :

Field , secret_hash_for_L1_to_l2_message :

Field , deadline_for_L1_to_l2_message :

Field , canceller_for_L1_to_L2_message :

EthAddress , caller_on_L1 :

EthAddress )

->

Field

{ let

mut hash_bytes :

[ u8 ;

324 ]

```
=

[ 0 ;

324 ] ;

// 10 fields of 32 bytes each + 4 bytes fn selector

let input_token_portal_bytes = input_asset_bridge_portal_address . to_field ( ) . to_be_bytes ( 32 ) ; let in_amount_bytes =
input_amount . to_be_bytes ( 32 ) ; let uniswap_fee_tier_bytes = uniswap_fee_tier . to_be_bytes ( 32 ) ; let
output_token_portal_bytes = output_asset_bridge_portal_address . to_field ( ) . to_be_bytes ( 32 ) ; let
amount_out_min_bytes = minimum_output_amount . to_be_bytes ( 32 ) ; let
secret_hash_for_redeeming_minted_notes_bytes = secret_hash_for_redeeming_minted_notes . to_be_bytes ( 32 ) ; let
secret_hash_for_L1_to_l2_message_bytes = secret_hash_for_L1_to_l2_message . to_be_bytes ( 32 ) ; let
deadline_for_L1_to_l2_message_bytes = deadline_for_L1_to_l2_message . to_be_bytes ( 32 ) ; let canceller_bytes =
canceller_for_L1_to_L2_message . to_field ( ) . to_be_bytes ( 32 ) ; let caller_on_L1_bytes = caller_on_L1 . to_field ( ) .
to_be_bytes ( 32 ) ;

// function selector: 0xbd87d14b
keccak256("swap_private(address,uint256,uint24,address,uint256,bytes32,bytes32,uint32,address,address)") hash_bytes [
0 ]

=

0xbd ; hash_bytes [ 1 ]

=

0x87 ; hash_bytes [ 2 ]

=

0xd1 ; hash_bytes [ 3 ]

=

0x4b ;

for i in

0 .. 32

{ hash_bytes [ i +

4 ]

= input_token_portal_bytes [ i ] ; hash_bytes [ i +

36 ]

= in_amount_bytes [ i ] ; hash_bytes [ i +

68 ]

= uniswap_fee_tier_bytes [ i ] ; hash_bytes [ i +

100 ]

= output_token_portal_bytes [ i ] ; hash_bytes [ i +

132 ]

= amount_out_min_bytes [ i ] ; hash_bytes [ i +

164 ]

= secret_hash_for_redeeming_minted_notes_bytes [ i ] ; hash_bytes [ i +

196 ]

= secret_hash_for_L1_to_l2_message_bytes [ i ] ; hash_bytes [ i +

228 ]
```

= deadline_for_L1_to_l2_message_bytes [ i ] ; hash_bytes [ i +

260 ]

= canceller_bytes [ i ] ; hash_bytes [ i +

292 ]

= caller_on_L1_bytes [ i ] ; } let content_hash =

sha256_to_field ( hash_bytes ) ; content_hash }Source code: noir-projects/noir-contracts/contracts/uniswap_contract/src/util.nr#L56-L105 This flow works similarly to the public flow with a few notable changes:

- Notice how in theswap_private()
- , user has to pass intoken
- address which they didn't in the public flow? Sinceswap_private()
- is a private method, it can't read what token is publicly stored on the token bridge, 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. Note that because public functions are executed by the sequencer while private methods are executed locally, all public calls are always done after all private calls are done. So first the burn would happen and only later the sequencer asserts that the token is same. Note that 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.
- In the public flow, the user callstransfer_public()
- . Here instead, the user callsunshield()
- . Why? The user can't directly transfer their private tokens, their notes to the uniswap contract, because later the Uniswap contract has to approve the bridge to burn these notes and withdraw to L1. The authwit flow for the private domain requires a signature from thesender
- , which in this case would be the Uniswap contract. For the contract to sign, it would need a private key associated to it. But who would operate this key?
- To work around this, the user can unshield their private tokens into Uniswap L2 contract. Unshielding would convert user's private notes to public balance. It is a private method on the token contract that reduces a user’s private balance and then calls a public method to increase the recipient’s (ie Uniswap) public balance.Remember that first all private methods are executed and then later all public methods will be - so the Uniswap contract won’t have the funds until public execution begins.
- Now uniswap has public balance (like with the public flow). Hence,swap_private()
- calls the internal public method which approves the input token bridge to burn Uniswap’s tokens and callsexit_to_l1_public
- to create an L2 → L1 message to exit to L1.
- Constructing the message content for swapping works exactly as the public flow except instead of specifying who would be the Aztec address that receives the swapped funds, we specify a secret hash (secret_hash_for_redeeming_minted_notes
- ). Only those who know the preimage to the secret can later redeem the minted notes to themselves.

In the next step we will write the code to execute this swap on L1Edit this page

Previous Solidity Code to Execute Swap on L1 Next Executing Private Swap on L1