How to write an accounts contract

This tutorial will take you through the process of writing your own account contract in Aztec.nr, along with the Typescript glue code required for using it within a wallet.

You will learn:

- How to write a custom account contract in Aztec.nr
- The entrypoint function for transaction authentication and call execution
- The AccountActions module and entrypoint payload structs, necessary inclusions for any account contract
- Customizing authorization validation within theis_valid
- function (using Schnorr signatures as an example)
- Typescript glue code to format and authenticate transactions
- Deploying and testing the account contract

Writing your own account contract allows you to define the rules by which user transactions are authorized and paid for, as well as how user keys are managed (including key rotation and recovery). In other words, writing an account contract lets you make the most out ofaccount abstraction in the Aztec network.

It is highly recommended that you understand how anaccount is defined in Aztec, as well as the differences between privacy and authenticationkeys . You will also need to know how to write acontract in Noir , as well as some basicTypescript .

For this tutorial, we will write an account contract that uses Schnorr signatures for authenticating transaction requests.

Every time a transaction payload is passed to this account contract'sentrypoint function, the account contract requires a valid Schnorr signature, whose signed message matches the transaction payload, and whose signer matches the account contract owner's public key. If the signature fails, the transaction will fail.

For the sake of simplicity, we will hardcode the signing public key into the contract, but you could store it in a private note ,in an immutable note , oron a separate keystore , to mention a few examples.

The account contract

Let's start with the account contract itself in Aztec.nr. Createa new Aztec.nr contract project that will contain a file with the code for the account contract, with a hardcoded public key:

contract // Account contract that uses Schnorr signatures for authentication using a hardcoded public key. contract SchnorrHardcodedAccount

{ use

dep :: std ; use

dep :: aztec :: prelude :: { AztecAddress ,

FunctionSelector ,

PrivateContext } ;

use

dep :: authwit :: { entrypoint :: { app :: AppPayload ,

fee :: FeePayload } ,

account :: AccountActions , auth_witness :: get_auth_witness } ;

global public_key_x :

Field

=

0x0ede3d33c920df8fdf43f3e39ed38b0882c25b056620ef52fd016fe811aa2443 ; global public_key_y :

Field

=

0x29155934ffaa105323695b5f91faadd84acc21f4a8bda2fad760f992d692bc7f ;

global ACCOUNT_ACTIONS_STORAGE_SLOT

=

1 ;

// Note: If you globally change the entrypoint signature don't forget to update default_entrypoint.ts

[aztec(private)]

fn

entrypoint (app_payload :

pub

AppPayload , fee_payload :

pub

FeePayload)

{ let actions =

AccountActions :: private (& mut context ,

ACCOUNT_ACTIONS_STORAGE_SLOT , is_valid_impl) ; actions . entrypoint (app_payload , fee_payload) ; }

[aztec(private)]

fn

spend_private_authwit (inner_hash :

Field)

->

Field

{ let actions =

AccountActions :: private (& mut context ,

ACCOUNT_ACTIONS_STORAGE_SLOT , is_valid_impl) ; actions . spend_private_authwit (inner_hash) }

[aztec(public)]

fn

spend_public_authwit (inner_hash :

Field)

->

Field

{ let actions =

AccountActions :: public (& mut context ,

ACCOUNT_ACTIONS_STORAGE_SLOT , is_valid_impl) ; actions . spend_public_authwit (inner_hash) }

[aztec(private)]

[aztec(internal)]

fn

cancel_authwit (outer_hash :

Field)

{ context . push_new_nullifier (outer_hash ,

0) ; }

[aztec(public)]

[aztec(internal)]

fn

approve_public_authwit (outer_hash :

Field)

{ let actions =

AccountActions :: public (& mut context ,

ACCOUNT_ACTIONS_STORAGE_SLOT , is_valid_impl) ; actions . approve_public_authwit (outer_hash) }

[contract_library_method]

fn

is_valid_impl (_context :

& mut

PrivateContext , outer_hash :

Field)

->

bool

{ // Load auth witness and format as an u8 array let witness :

[Field ;

64]

=

get_auth_witness (outer_hash) ; let

mut signature :

[u8 ;

64]

=

[0 ;

64] ; for i in

0 .. 64

{ signature [i]

= witness [i]

as

u8 ; }

// Verify signature using hardcoded public key let verification =

std :: schnorr :: verify_signature (public_key_x , public_key_y , signature , outer_hash . to_be_bytes (32)) ; assert (verification ==

true) ; true } }Source code: noir-projects/noir-contracts/contracts/schnorr_hardcoded_account_contract/src/main.nr#L1-L71 info You can usethe Aztec CLI to generate a new keypair if you want to use a different one:

aztec-cli generate-private-key Private Key: 0xc06461a031058f116f087bc0161b11c039648eb47e03bad3eab089709bf9b8ae Public Key: 0x0ede151adaef1cfcc1b3e152ea39f00c5cda3f3857cef00decb049d283672dc713c0e184340407e796411f74b7383252f1406272b58fccad6fee203f8a6db474 The important part of this contract is theentrypoint function, which will be the first function executed in any transaction originated from this account. This function has two main responsibilities: authenticating the transaction and executing calls. It receives apayload with the list of function calls to execute, and requests a corresponding auth witness from an oracle to validate it. You will find this logic implemented in theAccountActions module, which use theAppPayload andFeePayload structs:

entrypoint pub

fn

entrypoint (self , app_payload :

AppPayload , fee_payload :

FeePayload)

{ let valid_fn =

self . is_valid_impl ; let

mut private_context =

self . context . private . unwrap ( ) ;

let fee_hash = fee_payload . hash ( ) ; assert (valid_fn (private_context , fee_hash)) ; fee_payload . execute_calls (private_context) ; private_context . capture_min_revertible_side_effect_counter ( ) ;

let app_hash = app_payload . hash ( ) ; assert (valid_fn (private_context , app_hash)) ; app_payload . execute_calls (private_context) }Source code: noir-projects/aztec-nr/authwit/src/account.nr#L57-L71 app-payload-struct struct

AppPayload

{ function_calls :

[FunctionCall ;

ACCOUNT_MAX_CALLS] , nonce :

Field , } Source code: noir-projects/aztec-nr/authwit/src/entrypoint/app.nr#L14-L19 fee-payload-struct struct

FeePayload

{ function_calls :

[FunctionCall ;

MAX_FEE_FUNCTION_CALLS] , nonce :

Field , } [Source code: noir-projects/aztec-nr/authwit/src/entrypoint/fee.nr#L13-L18]() info Using theAccountActions module and the payload structs is not mandatory. You can package the instructions to be carried out by your account contract however you want. However, using these modules can save you a lot of time when writing a new account contract, both in Noir and in Typescript. TheAccountActions module provides default implementations for most of the account contract methods needed, but it requires a function for validating an auth witness. In this function you will customize how your account validates an action: whether it is using a specific signature scheme, a multi-party approval, a password, etc.

is-valid

[contract_library_method]

fn

is_valid_impl (_context :

& mut

PrivateContext , outer_hash :

Field)

->

bool

{ // Load auth witness and format as an u8 array let witness :

[Field ;

64]

=

get_auth_witness (outer_hash) ; let

mut signature :

[u8 ;

64]

=

[0 ;

64] ; for i in

0 .. 64

{ signature [i]

= witness [i]

as

u8 ; }

// Verify signature using hardcoded public key let verification =

std :: schnorr :: verify_signature (public_key_x , public_key_y , signature , outer_hash . to_be_bytes (32)) ; assert (verification ==

true) ; true } [Source code: noir-projects/noir-contracts/contracts/schnorr_hardcoded_account_contract/src/main.nr#L49-L69]() For our account contract, we will take the hash of the action to authorize, request the corresponding auth witness from the oracle, and validate it against our hardcoded public key. If the signature is correct, we authorize the action.

The typescript side of things

Now that we have a valid account contract, we need to write the typescript glue code that will take care of formatting and authenticating transactions so they can be processed by our contract, as well as deploying the contract during account setup. This takes the form of implementing theAccountContract interface from@aztec/aztec.js :

account-contract-interface /* ** An account contract instance. Knows its artifact, deployment arguments, how to create * transaction execution requests out of function calls, and how to authorize actions.* / export

interface

AccountContract

{ /* ** Returns the artifact of this account contract.*/ getContractArtifact ( ) : ContractArtifact ;

/* ** Returns the deployment arguments for this instance, or undefined if this contract does not require deployment*/ getDeploymentArgs ( ) :

any [ ]

|

undefined ;

/* * *Returns the account interface for this account contract given a deployment at the provided address. * The account interface is responsible for assembling tx requests given requested function calls, and * for creating signed auth witnesses given action identifiers (message hashes). * @param address - Address where this account contract is deployed. * @param nodeInfo - Info on the chain where it is deployed. * @returns An account interface instance for creating tx requests and authorizing actions.* / getInterface (address : CompleteAddress , nodeInfo : NodeInfo) : AccountInterface ; }Source code: yarn-project/aztec.js/src/account/contract.ts#L7-L33 However, if you are using the defaultAccountActions module, then you can leverage theDefaultAccountContract class from@aztec/accounts and just implement the logic for generating an auth witness that matches the one you wrote in Noir:

account-contract const

PRIVATE_KEY

= GrumpkinScalar . fromString ('0xd35d743ac0dfe3d6dbe6be8c877cb524a00ab1e3d52d7bada095dfc8894ccfa') ;

/* *Account contract implementation that authenticates txs using Schnorr signatures.*/ class

SchnorrHardcodedKeyAccountContract

extends

DefaultAccountContract

{ constructor (private privateKey : GrumpkinPrivateKey =

PRIVATE_KEY)

{ super (SchnorrHardcodedAccountContractArtifact) ; }

getDeploymentArgs ( ) :

undefined

{ // This contract has no constructor return

undefined ; }

getAuthWitnessProvider (_address : CompleteAddress) : AuthWitnessProvider { const privateKey =

this . privateKey ; return

{ createAuthWitness (message : Fr) :

Promise < AuthWitness

{ const signer =

new

Schnorr ( ) ; const signature = signer . constructSignature (message . toBuffer ( ) , privateKey) ; return

Promise . resolve (new

AuthWitness (message ,

[... signature . toBuffer ( )])) ; } , } ; } }Source code: yarn-project/end-to-end/src/guides/writing_an_account_contract.test.ts#L20-L45 As you can see in the snippet above, to fill in this base class, we need to define three things:

- The build artifact for the corresponding account contract.
- The deployment arguments.
- How to create an auth witness.

In our case, the auth witness will be generated by Schnorr-signing over the message identifier using the hardcoded key. To do this, we are using theSchnorr signer from the@aztec/circuits.js package to sign over the payload hash. This signer maps to exactly the same signing scheme that Noir's standard library expects inschnorr::verify_signature .

info More signing schemes are available in case you want to experiment with other types of keys. Check out Noir'sdocumentation on cryptographic primitives .

Trying it out

Let's try creating a new account backed by our account contract, and interact with a simple token contract to test it works.

To create and deploy the account, we will use theAccountManager class, which takes an instance of an Private Execution Environment (PXE), aprivacy private key , and an instance of ourAccountContract class:

account-contract-deploy const encryptionPrivateKey = GrumpkinScalar . random ( ) ; const account =

new

AccountManager (pxe , encryptionPrivateKey ,

new

SchnorrHardcodedKeyAccountContract ( )) ; const wallet =

await account . waitSetup ( ) ; const address = wallet . getCompleteAddress ( ) . address Source code: yarn-project/end-to-end/src/guides/writing_an_account_contract.test.ts#L58-L63 Note that we get a Wallet instance out of the account, which we can use for initializing the token contract class after deployment, so any transactions sent to it are sent from our wallet. We can then send a transaction to it and check its effects:

account-contract-works const token =

await TokenContract . deploy (wallet ,

{ address } ,

'TokenName' ,

'TokenSymbol' ,

18) . send ( ) . deployed ( ) ; logger (Deployed token contract at { token . address }) ;

const secret = Fr . random ( ) ; const secretHash =

computeMessageSecretHash (secret) ;

const mintAmount =

50n ; const receipt =

await token . methods . mint_private (mintAmount , secretHash) . send ( ) . wait ( ) ;

const storageSlot =

new

Fr (5) ; const noteTypeId =

new

Fr (84114971101151129711410111011678111116101n) ;

// TransparentNote

const note =

new

Note ([new

Fr (mintAmount) , secretHash]) ; const extendedNote =

new

ExtendedNote (note , address , token . address , storageSlot , noteTypeId , receipt . txHash) ; await pxe . addNote (extendedNote) ;

await token . methods . redeem_shield ({ address } , mintAmount , secret) . send ( ) . wait ( ) ;

const balance =

await token . methods . balance_of_private ({ address }) . view ( ) ; logger (Balance of wallet is now { balance }) ; Source code: yarn-project/end-to-end/src/guides/writing_an_account_contract.test.ts#L66-L87 If we run this, we get Balance of wallet is now 150 , which shows that the mint call was successfully executed from our account contract.

To make sure that we are actually validating the provided signature in our account contract, we can try signing with a different key. To do this, we will set up a new Account instance pointing to the contract we already deployed but using a wrong signing key:

account-contract-fails const wrongKey = GrumpkinScalar . random ( ) ; const wrongAccountContract =

new

SchnorrHardcodedKeyAccountContract (wrongKey) ; const wrongAccount =

new

AccountManager (pxe , encryptionPrivateKey , wrongAccountContract , account . salt) ; const wrongWallet =

await wrongAccount . getWallet ( ) ; const tokenWithWrongWallet = token . withWallet (wrongWallet) ;

try

{ await tokenWithWrongWallet . methods . mint_private (200 , secretHash) . simulate ( ) ; }

catch

(err)

{ logger (Failed to send tx: { err }) ; } Source code: yarn-project/end-to-end/src/guides/writing_an_account_contract.test.ts#L90-L102 Lo and behold, we get Error: Assertion failed: 'verification == true' when running the snippet above, pointing to the line in our account contract where we verify the Schnorr signature. Edit this page

Previous Storage slots Next How to emit an event