Keys

Typically, each account in Aztec is backed by two separate keys:

- Asigning key
- used for authenticating the owner of the account.
- Aprivacy master key
- used for deriving encryption and nullifying keys for managing private state.

Signing keys

Signing keys allow their holder to act as their corresponding account in Aztec, similarly to the keys used for an Ethereum account. If a signing key is leaked, the user can potentially lose all their funds.

Since Aztec implements fullsignature abstraction , signing keys depend on the account contract implementation for each user. Usually, an account contract will validate a signature of the incoming payload against a known public key.

This is a snippet of our Schnorr Account contract implementation, which uses Schnorr signatures for authentication:

entrypoint // Load public key from storage let storage =

Storage :: init (Context :: private (context)) ; let public_key = storage . signing_public_key . get_note ( ) ; // Load auth witness 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 of the payload bytes let verification =

std :: schnorr :: verify_signature (public_key . x , public_key . y , signature , outer_hash . to_be_bytes (32)) ; assert (verification ==

true) ; Source code: noir-projects/noir-contracts/contracts/schnorr_account_contract/src/main.nr#L71-L92 Still, different accounts may use different signing schemes, may require multi-factor authentication, ormay not even use signing keys and instead rely on other authentication mechanisms. Readhow to write an account contract for a full example of how to manage authentication.

Furthermore, and since signatures are fully abstracted, how the key is stored in the contract is abstracted as well and left to the developer of the account contract. Here are a few ideas on how to store them, each with their pros and cons.

Using a private note

Storing the signing public key in a private note makes it accessible from the entrypoint function, which is required to be a

private function, and allows for rotating the key when needed. However, keep in mind that reading a private note requires nullifying it to ensure it is up to date, so each transaction you send will destroy and recreate the public key. This has the side effect of enforcing a strict ordering across all transactions, since each transaction will refer the instantiation of the private note from the previous one.

Using an immutable private note

Similar to using a private note, but using an immutable private note removes the need to nullify the note on every read. This generates less nullifiers and commitments per transaction, and does not enforce an order across transactions. However, it does not allow the user to rotate their key should they lose it.

Using the slow updates tree

A compromise between the two solutions above is to use theslow updates tree . This would not generate additional nullifiers and commitments for each transaction while allowing the user to rotate their key. However, this causes every transaction to now have a time-to-live determined by the frequency of the slow updates tree.

Reusing the privacy master key

It is possible to use the privacy master key as the signing key also. Since this key is part of the address preimage (more on this on the privacy master key section), you can validate it against the account contract address rather than having to store it. However, this approach is not recommended since it reduces the security of the user's account.

Using a separate keystore

Since there are no restrictions on the actions that an account contract may execute for authenticating a transaction (as long as these are all private function executions), the signing public keys can be stored in aseparate keystore contract that is checked on every call. This will incur in a higher proving time for each transaction, but has no additional cost in terms of fees, and allows for easier key management in a centralized contract.

Privacy keys

Each account is tied to aprivacy master key . Unlike signing keys, privacy keys are enshrined at the protocol layer, are required to be Grumpkin keys, and are tied to their account address. These keys are used for deriving encryption and nullifying keys, scoped to each application, in a manner similar to BIP32.

danger At the time of this writing, privacy master keys are used by applications without any derivation whatsoever. This means that the private key is used directly as a nullifier secret for all applications, and the public key is used as an encryption key for all purposes. This is highly insecure, and will change to match the specification below in an upcoming release.

Addresses, partial addresses, and public keys

When deploying a contract, the address is deterministically derived from the contract code, the constructor arguments, a salt, and a public key:

partial_address := hash(salt, contract_code, constructor_hash) address := hash(public_key, partial_address) This public key corresponds to the privacy master key of the account. In order to manage private state, such as receiving an encrypted note, an account needs to share its partial address and public key, along with its address. This allows anyone to verify that the public key corresponds to the intended address. We call the address, partial address, and public key of a user theircomplete address .

Contracts that are not meant to represent a user who handles private state, usually non-account contracts such as applications, do not need to provide a valid public key, and can instead just use zero to denote that they are not expected to receive private notes.

info A side effect of enshrining and encoding privacy keys into the account address is that these keys cannot be rotated if they are leaked. Read more about this in theaccount abstraction section .

Encryption keys

The privacy master key is used to derive encryption keys. Encryption keys, as their name implies, are used for encrypting private notes for a recipient, where the public key is used for encryption and the corresponding private key used for decryption.

In a future version, encryption keys will be differentiated between incoming and outgoing. When sending a note to another user, the sender will use the recipient's incoming encryption key for encrypting the data for them, and will optionally use their own outgoing encryption key for encrypting any data about the destination of that note. This is useful for reconstructing transaction history from on-chain data. For example, during a token transfer, the token contract may dictate that the sender

encrypts the note with value with the recipient's incoming key, but also records the transfer with its own outgoing key for bookkeeping purposes.

An application in Aztec.nr can access the encryption public key for a given address using the oracle callget_public_key , which you can then use for calls such asemit_encrypted_log :

encrypted emit_encrypted_log (context , (* context) . this_address ( ) , slot , Self :: get_note_type_id ( ) , encryption_pub_key , self . serialize_content ( ) ,) ; Source code: noir-projects/aztec-nr/address-note/src/address_note.nr#L74-L83 info In order to be able to provide the public encryption key for a given address, that public key needs to have been registered in advance. At the moment, there is no broadcasting mechanism for public keys, which means that you will need to manually register all addresses you intend to send encrypted notes to. You can do this via theregisterRecipient method of the Private Execution Environment (PXE), callable either via aztec.js or the CLI. Note that any accounts you own that have been added to the PXE are automatically registered.

Nullifier secrets

In addition to deriving encryption keys, the privacy master key is used for deriving nullifier secrets. Whenever a private note is consumed, a nullifier deterministically derived from it is emitted. This mechanisms prevents double-spends, since nullifiers are checked by the protocol to be unique. Now, in order to preserve privacy, a third party should not be able to link a note commitment to its nullifier - this link is enforced by the note implementation. Therefore, calculating the nullifier for a note requires a secret from its owner.

An application in Aztec.nr can request a secret from the current user for computing the nullifier of a note via therequest_nullifier_secret_key api:

nullifier fn

compute_nullifier (self , context :

& mut

PrivateContext)

->

Field

{ let note_hash_for_nullify =

compute_note_hash_for_consumption (self) ; let secret = context . request_nullifier_secret_key (self . owner) ; // TODO(#1205) Should use a non-zero generator index. pedersen_hash ([note_hash_for_nullify , secret . low , secret . high ,] , 0) } Source code: noir-projects/aztec-nr/value-note/src/value_note.nr#L38-L51

Scoped keys

danger Keys are not yet scoped at the time of this writing. This will be implemented in a future release. Even though they are all derived from the same privacy master key, all encryption and nullifier keys are scoped to the contract that requests them. This means that the encryption key used for the same user in two different application contracts will be different. The same applies to nullifier secrets.

This allows per-application auditability. A user may choose to disclose their inbound and outbound encryption keys for a given application to an auditor or regulator, as a means to reveal all their activity within that context, while retaining privacy across all other applications in the network.

In the case of nullifier secrets, there is also a security reason involved. Since the nullifier secret is exposed in plain text to the application contract, the contract may accidentally or maliciously leak it. If that happens, only the nullifier secret for that application is compromised.

Security considerations

A leaked privacy master key means a loss of privacy for the affected user. An attacker who holds the privacy private key of a user can derive the encryption private keys to decrypt all past inbound and outbound private notes, and can derive the nullifier secrets to determine when these notes were consumed.

Nevertheless, the attacker cannot steal the affected user's funds, since authentication and access control depend on the signing keys and are managed by the user's account contract.

info Note that, in the current architecture, the user's wallet needs direct access to the privacy private key, since the wallet needs to use this key for attempting decryption of all notes potentially sent to the user. This means that the privacy private key cannot be stored in a hardware wallet or hardware security module, since the wallet software uses the private key material directly. This may change in future versions in order to enhance security. Edit this page