Wallets

In this page we will cover the main responsibilities of a wallet in the Aztec network.

Refer towriting an account contract for a tutorial on how to write a contract to back a user's account.

Go to_wallet architecture for an overview of its architecture and a reference on the interface a wallet must implement.

Wallets are the applications through which users manage their accounts. Users rely on wallets to browse through their accounts, monitor their balances, and create new accounts. Wallets also store seed phrases and private keys, or interact with external keystores such as hardware wallets.

Wallets also provide an interface for dapps. Dapps may request access to see the user accounts, in order to show the state of those accounts in the context of the application, and request to send transactions from those accounts as the user interacts with the dapp.

In addition to these usual responsibilities, wallets in Aztec also need to track private state. This implies keeping a local database of all private notes encrypted for any of the user's accounts, so dapps and contracts can query the user's private state. Aztec wallets are also responsible for producing local proofs of execution for private functions.

Account setup

The first step for any wallet is to let the user set up theiraccounts . An account in Aztec is represented on-chain by its corresponding account contract that the user must deploy to begin interacting with the network. This account contract dictates how transactions are authenticated and executed.

A wallet must support at least one specificaccount contract implementation , which means being able to deploy such a contract, as well as interacting with it when sending transactions. Code-wise, this requiresimplementing theAccountContract interface .

Note that users must be able to receive funds in Aztec before deploying their account. A wallet should let a user generate adeterministic complete address without having to interact with the network, so they can share it with others to receive funds. This requires that the wallet pins a specific contract implementation, its initialization arguments, a deployment salt, and a privacy key. These values yield a deterministic address, so when the account contract is actually deployed, it is available at the precalculated address. Once the account contract is deployed, the user can start sending transactions using it as the transaction origin.

Transaction lifecycle

Every transaction in Aztec is broadcast to the network as a zero-knowledge proof of correct execution, in order to preserve privacy. This means that transaction proofs are generated on the wallet and not on a remote node. This is one of the biggest differences with regard to EVM chain wallets.

A wallet is responsible forcreating anexecution request out of one or morefunction calls requested by a dapp. For example, a dapp may request a wallet to "invoke thetransfer function on the contract at0x1234 with the following arguments", in response to a user action. The walletturns that into an execution request with the signed instructions to execute that function call from the user's account contract. In anECDSA-based account , for instance, this is an execution request that encodes the function call in theentrypoint payload , and includes its ECDSA signature with the account's signing private key.

Once theexecution request is created, the wallet is responsible forsimulating andproving the execution of its private functions. The simulation yields an execution trace, which can be used to provide the user with a list of side effects of the private execution of the transaction. During this simulation, the wallet is responsible of providing data to the virtual machine, such as private notes, encryption keys, or nullifier secrets. This execution trace is fed into the prover, which returns a zero-knowledge proof that guarantees correct execution and hides all private information. The output of this process is atransaction object.

info Since private functions rely on a UTXO model, the private execution trace of a transaction is determined exclusively by the notes used as inputs. Since these notes are immutable, the trace of a transaction is always the same, so any effects observed during simulation will be exactly the same when the transaction is mined. However, the transaction may be dropped if it attempts to consume a private note that another transaction nullified before it gets mined. Note that this applies only to private function execution. Public functions rely on an account model, similar to Ethereum, so their execution trace depends on the chain's public state at the point they are included in a block, which may have changed since the transaction was simulated locally. Finally, the walletsends the resultingtransaction object, which includes the proof of execution, to an Aztec Node. The transaction is then broadcasted through the peer-to-peer network, to be eventually picked up by a sequencer and included in a block.

danger There are no proofs generated as of the Sandbox release. This will be included in a future release before testnet.

Authorizing actions

Account contracts in Aztec expose an interface for other contracts to validate whether an action is authorized by the account or not . For example, an application contract may want to transfer tokens on behalf of a user, in which case the token contract will check with the account contract whether the application is authorized to do so. These actions may be carried out in private or in public functions, and in transactions originated by the user or by someone else.

Wallets should manage these authorizations, prompting the user when they are requested by an application. Authorizations in private executions come in the form ofauth witnesses , which are usually signatures over an identifier for an action. Applications can request the wallet to produce an auth witness via thecreateAuthWitness call. In public functions, authorizations are pre-stored in the account contract storage, which is handled by a call to an internal function in the account contract implementation.

Key management

As in EVM-based chains, wallets are expected to manage user keys, or provide an interface to hardware wallets or alternative key stores. Keep in mind that in Aztec each account requirestwo sets of keys : privacy keys and authentication keys. Privacy keys are mandated by the protocol and used for encryption and nullification, whereas authentication keys are dependent on the account contract implementation rolled out by the wallet. Should the account contract support it, wallets must provide the user with the means to rotate or recover their authentication keys.

info Due to limitations in the current architecture, privacy keys need to be available in the wallet software itself and cannot be punted to an external keystore. This restriction may be lifted in a future release.

Recipient encryption keys

Wallets are also expected to manage the public encryption keys of any recipients of local transactions. When creating an encrypted note for a recipient given their address, the wallet needs to provide theircomplete address . Recipients broadcast their complete addresses when deploying their account contracts, and wallets collect this information and save it in a local registry for easy access when needed.

Note that, in order to interact with a recipient who has not yet deployed their account contract (and thus not broadcasted their complete address), it must also be possible to manually add an entry to a wallet's local registry of complete addresses.

Private state

Last but not least, wallets also store the user's private state. Wallets currently rely on brute force decryption, where every new block is downloaded and its encrypted data blobs are attempted to be decrypted with the user decryption keys. Whenever a blob is decrypted properly, it is added to the corresponding account's private state. Note that wallets must also scan for private state in blocks prior to the deployment of a user's account contract, since users may have received private state before deployment.

info At the time of this writing, all private state is encrypted and broadcasted through the network, and eventually committed to L1. This means that a wallet can reconstruct its entire private state out of its encryption keys in the event of local data loss. Encrypted data blobs do not carry any public information as to whom their recipient is. Therefore, it is not possible for a remote node to identify the notes that belong to a user, and it is not possible for a wallet to query a remote node for its private state. As such, wallets need to keep a local database of their accounts private state, in order to be able to answer any queries on their private state.

Dapps may require access to the user's private state, in order to show information relevant to the current application. For instance, a dapp for a token may require access to the user's private notes in the token contract in order to display the user's balance. It is responsibility of the wallet to require authorization from the user before disclosing private state to a dapp. Edit this page

Previous Versions and Updating Next Architecture