TLDR: With the introduction of auth witnesses, we now have an easy path to simulating simulations in a wallet, so we can check what authorisations are required by a tx with an initial simulation, validate those with the user, and then proceed with the actual simulation and proof generation.

About auth witnesses

Auth witnesses, originally devised by @LHerskind here, are a mechanism for account contracts to request authorisation for actions from the oracle. Account contracts now expose an is_valid

function, that receives a message hash that represents an action. Within is_valid

, account contracts are expected to request a corresponding auth witness

from the oracle, which is typically a signature over the message hash, and validate it according to its own validation rules. For instance, an ECDSA contract does something like this:

fn is_valid(context: &mut PrivateContext, payload_hash: Field) -> pub bool { // Load public key from storage let storage = Storage::init(Context::private(context)); let public_key = storage.public_key.get_note();

```
// Get the auth witness from the oracle
let signature: [Field; 64] = get_auth_witness(payload_hash);

// Verify payload signature using Ethereum's signing scheme
let hashed_message: [u8; 32] = std::hash::sha256(payload_hash.to_be_bytes(32));
let verification = std::ecdsa_secp256k1::verify_signature(public_key, signature, hashed_message);
assert(verification == true);
```

}

This mechanism is used for authorising function calls done through the account contract (ie the EntrypointPayload

), but also for authorising any actions requested by another contract in a transaction. For instance, this is used in token contracts as a replacement for ERC20-like approvals. If a token contract needs to run a transfer_from

, it just asks the sender whether that transfer is_valid

or not.

Auth witnesses are only valid in private execution contexts. For public functions, we rely on setting in the account's public storage that a message hash is authorised, so calls to is_valid_public

can just check storage.

Note that we could change public functions to also rely on auth witnesses if we make the sequencer aware of them, and broadcast them along with a tx, but this requires a protocol change.

All this means that a wallet needs to generate all private auth witnesses prior to locally executing a transaction, and needs to set all public approvals prior to calling the public functions that will rely on them.

So how does a wallet know what actions will need to be authorised?

Collecting authorisation requests

Given that all authorisations are now condensed in a single function in the account contract, a wallet can initially run an execution request in a modified version of the account contract where is_valid

just returns true and records all requests it received. This would be effectively simulating the execution simulation, and the wallet would end up with the list of all actions (entrypoint included) that need to be authorised by the user.

This means that the flow for sending a tx would now be:

- Dapp tells the wallet call token.transfer(...)

on behalf of the user

- Wallet packages the call in an EntrypointPayload
- Wallet simulates the entrypoint

function using the modified account

that accepts all actions

- Given the list of actions to be authorised, both private and public, the wallet asks the user to confirm
- After the user agrees, the wallet produces all auth witnesses and sets public authorisations (as part of the transaction), and runs the simulation with the actual account contract
- Wallet produces a proof out of the actual simulation and broadcasts the tx

Getting authorisation preimages

A complication with the scheme above is that the is_valid

function receives a hash

of the action to authorise. And we cannot show the user a hash, they need to see the preimage to understand what they are signing.

I can think of a few options around this:

- We modify is_valid

so that it receives the raw payload and not the hash. I'm not sure if this could be an issue when it comes to large payloads, especially in public-land. This may also not work if the hash is calculated in advance in a prior tx (eg as it happens in some multisig or timelock contracts, where the hash of the action is stored, but the actual action isn't).

- We instrument all calls to hash functions so that we store all hash operations, and can retrieve the preimages for a given value. This feels brittle, and it fails if a contract happens to tweak the hash (eg by XORing it with another value) in any way. It also fails in the case of pre-stored hashes.
- We require any calling contracts to execute an oracle call where they emit the preimage right before calling into is_valid

. This oracle call would be fully unconstrained, and would be used just to provide a hint to the wallet on what info to display to the user. This would require making oracle calls from public functions, that are only executed in local simulations but discarded when run by a sequencer. If a hint is not provided for a given is_valid

call, wallets can outright refuse to run the tx, or tell their users they are approving an operation blindly.

Making sense of preimages

Assuming we can get our hands on authorisation preimages, next step is to actually make sense of them. Each application would package an authorisation payload however it makes sense to them, but we need wallets to be able to translate this into something user-friendlish.

One option is to use EIP712, where we encode the structure and domain separator of the data into its hash. Or we can use an events-like signature for these preimage hints, where contracts declare in their ABI the structure of their auth requests (as they declare the structure of their emitted events), and wallets use this for deserialising the requests and presenting them to their users.

Altered auth request hashes in public-land

Another problem is that the auth payload may change when being executed by the sequencer. If wallets collect auth request hashes during a local simulation of the public part of a tx, the execution path may change when it is run by the sequencer (since the public state of the chain may be different by that time). This means that if an auth request depends on a value that may change, the resulting auth request hash will change, and the is_valid

check would fail. This can happen for instance when authorising a token transfer, where the amount to be transferred depends on the state of the chain.

Supporting this may require to include in the public function call the exact values used to construct the auth witness. So, in the token transfer example, the public function could be called with a "up to X tokens are authorised", so the contract assembles the auth request using the exact X value. However, this requires devs to be aware of this limitation and makes the flow more complex overall.

Prototyping this in the Sandbox

It’d be good to start prototyping this on top of the Sandbox. Today, the role of a Wallet is split across the client and the sandbox: the CLI or aztec.js takes care of assembling and signing txs, while the Sandbox runs the RPC Server (or Private Execution Environment?).

If we continue with this approach, we would need to implement this new authorisation flow on the client, where auth witnesses can be generate, and where we can reach out to the user to ask for confirmations. However, this would require extending the RPC Server interface so the client can request simulations with a modified account contract, and collect the is_valid

requests somehow.

Alternatively, we can break up the Sandbox into Wallet and Node, more closely resembling how a testnet would look like, and have this implemented into the Wallet itself. But this is a much larger architecture change for local development.