The Wonderland team is building a [private oracle

contract](https://github.com/defi-wonderland/aztec-private-oracle/) on the Sandbox. This contract allows any user to ask a question

privately to a divinity

of their choice, locking down a fee

in a given Token

. The divinity

can then provide an answer

for a question

, which unlocks the fee payment to them, or the user can cancel

their question, which returns the fee

to them. Both answering or cancelling a question emit the same nullifier, ensuring only one of those actions is possible.

The problem comes when locking the fee

in the Oracle

contract. This is achieved by transferring the funds, via an authwitted private token.transfer

call, from the user to the Oracle

contract. However, the Oracle

contract, being an “app” contract, doesn’t have a private key

, so it cannot receive private value notes. So how do we deal with this…?

Using a “public” encryption private key for the contract

We deploy the Oracle

contract with a private key that’s shared with the world. Anyone who wants to interact with the contract, simply registers that key in their pxe so they can “see” all the value notes owned by the oracle. This is the simplest solution, but it leaks privacy, since now everyone can see when a private tx calls the oracle contract.

Using public token balances

Another easy solution: instead of calling token.transfer

in the Oracle

contract, we unshield

from the user to it, and then shield

back to the user or to the divinity. This leaks not just that the Oracle

is in use, but also who the answering divinity is in each transfer.

Escrow contract per question (or per user/divinity pair)

Whenever we need to escrow funds in a note that must be visible by a user and a divinity, we deploy a new contract with an encryption key that’s shared between the two. The shared encryption key can be generated using the following scheme:

PubKeyAlice = PrivKeyAlice * G PubKeyBob = PrivKeyBob * G SS = PubKeyAlice * PrivKeyBob = PubKeyBob * PrivKeyAlice

Knowing the deployment pubkey and bytecode of the escrow contract, it should be possible to reconstruct the expected escrow contract address within the Oracle

contract (like the address of Uniswap trading pair contracts can be reconstructed from the address of each token).

Now, whenever we need to lock the fee from the user, we transfer it to a specific escrow contract with an encryption key that allows both the user and the divinity to “see” the value note and use it for either answering or cancelling the question.

However, this adds a massive overhead to the protocol. It also needs a separate transaction for deploying each escrow contract, since we don’t yet support deploying a contract from another one.

Transfer to a shared pubkey

The easiest solution would probably be to create a value note owned by the Oracle

contract but encrypted for a pubkey specific to the user/divinity pair. However, we don’t support that in our Token

contract implementation since we mapped addresses and pubkeys 1:1.

To support this, we would need to modify the token transfer

function (or add an overload) to receive the recipient address and pubkey separately. The Oracle

contract would then calculate the shared encryption private key using a combination of get_secret_key(user)

and get_public_key(divinity)

, derive the public key, and execute a transfer

to self using that derived pubkey.

Note that in both this scenario and the previous one the app would need to register this shared encryption key as a new account on both the user and the recipient pxe

s, which means a large additional note decryption overhead for a pretty much ephemeral pubkey, that can be tossed out once the question was finished.

Reencrypt the same note under multiple pubkeys

An alternative that does not require overloading the pxe

with additional keys is to simply re-encrypt the same note under the pubkey of each recipient. In other words, having a single commitment on the note hash tree, but emit it as an encrypted log multiple times. The tradeoff here is more calldata used in the tx.

An issue with this approach is nullifying the note. Since multiple users have “access” to the note, they all need to be able to nullify it. The shared secret, derived from the recipients pubkeys, could be used here. Or a random nullifier key could be just embedded as part of the encrypted note, which becomes visible to all those who can decrypt it.

This implies that we cannot use the same note implementation for multi-encryption that we use for individual transfers, since we cannot just rely on get_secret(note.owner)

for nullifying. But it’s a fairly small change, and it also plays well withstoring nullifier secrets or identifiers as part of the note itself if we allow for rotating nullifier keys.