Context

Can we leverage Portals and account abstraction to create some type of “L1 social recovery” system where in the event of issues, whether they’re bugs, hacks, exploits, etc., users can have a mechanism that enables their funds to be “exited” to L1 portals, where it can stay until the upgraded L2 is alive?

Requirements to satisfy are:

1. Users can grant permissions to a third party that can move their funds to the L1 portal users deposited into
2. The third party cannot do anything else with their permissions/privileges other than escape funds to the L1 portal users deposited into.
3. Users can add, change, or remove who the “recover-er” trusted third party is
4. It is reasonably cost-efficient (at least enough that services can charge for it)
5. It is reasonably efficient enough (perhaps with the usage ofmass portal migrations) to actually solve the issue (letting users escape and/or race more quickly than a potential attacker)
6. Handle funds both in the private state or public state.
7. Make it as minimal as possible on L2 as we don’t know what bugs may exist on L2!
8. Should be (ideally) extendable for any dapp (defi or otherwise) that has an L1 portal contract, and not just vanilla token contracts.

Summary

We consider two designs -

1. A generic recovery portal where the user’s funds will stay temporarily until the new L2 is out.
2. We don’t require anything from applications to support this.
3. just need to ensure that this recovery portal doesn’t have any bugs in its recovery implementation
4. However, at recovery time, the user would need to perform some interaction (e.g. provide authwitness detailing the amount to recover)
5. We don’t require anything from applications to support this.
6. just need to ensure that this recovery portal doesn’t have any bugs in its recovery implementation
7. However, at recovery time, the user would need to perform some interaction (e.g. provide authwitness detailing the amount to recover)
8. Every application supports its own form of recovery.
9. Each application has to do its own due diligence on their implementation
10. Possible without any user interaction at recovery time
11. In the private flow, the operator needs to know the user’s decryption key ahead of time (they will know this in (1) too but only at recovery time)
12. Each application has to do its own due diligence on their implementation
13. Possible without any user interaction at recovery time
14. In the private flow, the operator needs to know the user’s decryption key ahead of time (they will know this in (1) too but only at recovery time)

Both designs:

- are easily extendable to any kind of public state or custom notes as long as the applications have an L1 portal that has the functionality to deposit or withdraw state from L2.
- Let you temporarily keep state in L1 while L2 is upgraded and then ship everything back to L2
- Work with aggregation to amortise costs

- require users to perform actions individually for each app. This is much better in the 2nd design as the user can do this at any point during the use of Aztec!
- fail for bugs in
- rollup contracts that may break L1<>L2 message system
- auth witness flow - signature
- Pederson/keccak hash implementation which would be modified in the new L2

Glossary

- Token Portal - L1 contract assosciated to a ERC20 that is responsible for depositing assets to L2 or withdrawing back to L1
- Token Bridge - the L2 counterpart to a token portal that is responsible for depositing assets to L2 or withdrawing back to L1
- Rescuer/Operator - person/group of operators governed by a DAO, a company that carries out recovery for the user
- Recovery Contract - contract on L2 deployed by recover-er.
- Recovery Portal - L1 contract deployed by recoverer where funds can stay.

Design 1 - Generic Recovery Portal

We have a recovery contract on L2 with its portal on L1.

sequenceDiagram participant R as Rahul participant L as Lasse participant A as Recovery Contract participant B as Recovery Portal participant G as Governance on L1 participant Re as Registry participant TB as Token Bridge participant TP as Token Portal participant T2 as Token L2 participant T1 as Token L1

R-->>L: authWit for transfer by recovery

G->>Re: Yo, start upgrade delay

B->>Re: Is there a pending upgrade
Re->>B: Yes
B->>A: There is a pending upgrade 😱

TP->>Re: Is there a pending upgrade
Re->>TP: Yes
TP->>TB: There is a pending upgrade 😱

L->>A: Yo, I am here to save Rahul
A->>A: There is a pending upgrade, its allowed
A->>TB: This guy wanna leave
TB->>T2: Burn these tokens
T2->>L: Yo, are you allowed to do this?
L->>T2: Yes, here is the authwit
T2->>TB: Tokens are burned
TB->>TP: Free some tokens to Rahul
TP->>T1: Transfer tokens to Rahul

How to know on L2 if there is a pending upgrade

1. Recovery portal could call the registry to find out if an upgrade delay is in progress. (Anyone can call this function at any time).
2. If so, create a L1 to L2 message for its “sister” contract on L2 (recovery contract): hash("setRecoveryMode(true)")

On L2 - The recovery contract has a public storage variable - bool RECOVER_MODE

and a consumeRecoveryMessage()

callable by anyone to consume the corresponding L1 to L2 message.

The RECOVER_MODE

variable keeps the rescuer in check and only allows the rescuer to move funds only during this time.

“Registering” with a rescuer

Users can’t predict in advance how many tokens they would have to rescue. The way authwitness is calculated in individual contracts (such as token contracts), you need to provide a precise number and not some upper bound. Users could provide a lower bound, but if they don’t have that many tokens, recovery would fail.

So there is no benefit in registering in advance.

At rescue time

(Assume L2 Recovery Contract is already in RECOVER_MODE

.

On the buggy L2:

A diligent user hears that there might be an exploit. So they generate:

1. authwitness to allow l2 recovery contract to transfer user’s fund to itself (hash("transfer/unshield(amount, to)"

)

1. authwitness detailing rescue parameters:

rescueFunds(token_bridge_address, this_aztec_version, hash("mint_in_new_l2(amount, recipient)")

Operator then calls rescueFunds(token_bridge_address, amount,)

which:

1. transfer/unshield the user’s tokens to itself
2. Creates a custom note containing from

(user), amount

, l1TokenPortalAddress

, hash("mint_in_new_l2(amount, recipient)"

and computes the note hash. A Merkle root of existing note hashes is constructed.

1. The contract has an internal accounting of the total amount of each token that it is rescuing.
2. At any time (epoch), an operator can burn these funds on L2, exit to L1 and create an L2 to L1 message to detail recover parameters (such as the Merkle root of the note hashes).

On L1:

Operator would

1. Consume L2 to L1 messages (the merkle root)
2. Talk to each portal to mint the combined funds to the L1RecoveryContract.
3. A Merkle tree could be constructed with all the roots (from each epoch) to keep track of all the individual note hashes.

After L2 is upgraded:

On L1 -

1. send the aggregated Merkle root to the new L2.
2. For all of its token balances, it talks to individual token portals and creates a deposit message

This allows portals to deploy a brand new token contract if they wish on the new L2!

On L2:

operators would pass the preimage of the note, the Merkle proofs and the token bridge address to mint tokens to the users.

The Merkle tree would exist on L1 or another DA solution so that people have enough information to build Merkle proofs.

Issues and when it doesn’t work

1. At rescue time, the user has to be attentive
2. Breaks if bugs happen in
3. L1<>L2 communication,
4. hashing algorithm (Pederson, keccak - used in L1/L2 + computing note hash)
5. authwitness flow
6. L1<>L2 communication,
7. hashing algorithm (Pederson, keccak - used in L1/L2 + computing note hash)
8. authwitness flow
9. Not straightforward if the user simply wants to exit to L1. We might want to create two different Merkle trees (one for those who want to exit to L1, and one who want to be moved back to new L2).
10. Requires user's Aztec address on new L2 - requires the user to know their new L2 address. We could use a secret hash but what if the new L2 has a new Pederson hash generator?
11. The user has to take action per token/application

Pros

1. Doesnt require work from any applications to support this (as long as they have a portal that handles depositing/withdrawing of assets)
2. Works for both public and private flow
3. Extensible to any kind of state (e.g. custom notes, public state)

Design 2 - Each application on its own:

sequenceDiagram participant R as Rahul participant L as Lasse participant A as Recovery Contract participant B as Portal participant G as Governance on L1 participant Re as Registry participant TB as Token Bridge participant TP as Token Portal participant T2 as Token L2 participant T1 as Token L1

R-->>L: authWit for transfer by recovery

G->>Re: Yo, start upgrade delay

B->>Re: Is there a pending upgrade
Re->>B: Yes
B->>A: There is a pending upgrade 😱

L->>A: Yo, I am here to save Rahul
A->>A: There is a pending upgrade, its allowed
A->>TB: This guy wanna leave
TB->>T2: Burn these tokens
T2->>L: Yo, are you allowed to do this?
L->>T2: Yes, here is the authwit
T2->>TB: Tokens are burned
TB->>TP: Free some tokens to Rahul
TP->>T1: Transfer tokens to Rahul

To facilitate these, the token bridge and portal need to have some specific methods that we explain as we go:

How to know on L2 if there is a pending upgrade

1. The token portal could call the registry to find out if an upgrade delay is in progress. (Anyone can call this function at any time).
2. If so, create a L1 to L2 message for the bridge: hash("setRecoveryMode(true)")

On L2 - the Bridge contract has a public storage variable - bool RECOVER_MODE

and a consumeRecoveryMessage()

callable by anyone to consume the corresponding L1 to L2 message.

The RECOVER_MODE

variable keeps the rescuer in check and only allows the rescuer to move funds only during this time.

The L2 Token contract could also store this.

“Registering” with a rescuer

At any point during a user’s interaction on the L2, they would create

1. AuthWitness to allow the operator to transfer an upper bound amount of funds to a recovery contract: transferForRecovery(operator, upperBoundAmount
2. authwitness detailing rescue parameters:

rescueFunds(token_bridge_address, this_aztec_version, hash("mint_in_new_l2(amount, recipient)")

Upperbound amount because we don’t know the amount to rescue in advance. This can work since each application is handling its own recovery process i.e. it is in their scope to modify the token contract!

At rescue time

Assume L2 is already in RECOVER_MODE

.

On buggy L2

The operator transfers the user’s funds to the recovery contract. Note that the operator only has an upper bound allowance. The token knows it is in “recovery mode” and that the transfer authwitness is not a generic transfer

but instead transferForRecovery

, so it lets the user transfer max(user_balance, upper bound)

to the recovery contract.

If the operator knows the user’s decryption key, then they can do this exact trick for the private state too (except they would unshield instead of transfer)

The operator now calls the recovery contract which would

1. call the bridge to burn its own assets and exit to L1.
2. Create a L2 to L1 message: "recoverFromL2(amount, hash("mint_in_new_l2(amount, recipient)"))"

.

Anyone can call the corresponding method on L1 to get the l1RecoveryPortal to consume this.

Note - if the user wishes to simply exit to L1 instead of using the new upgraded L2 (whenever it would be deployed), they can pass that instead of hash("mint_in_new_l2(amount, recipient)")

On L1:

As 3rd party:

1. Consumes the withdraw message from the token portal to transfer tokens to the l1RecoveryPortal
2. mint user’s tokens to l1RecoveryPortal
3. consume the recoverFromL2

L2 to L1 message

1. create a Merkle tree of the addresses and amounts (similar to the Merkle tree airdrop idea suggested in mass migrations).

After L2 is upgraded:

On L1

- send merkle root to L2.

- Talk to individual token portals to create a deposit message back to L2

Issues and when it doesn't work

1. Expensive - although aggregation can be used as explained in Design 1
2. Breaks if bugs happen in
3. L1<>L2 communication,
4. hashing algorithm (Pederson, keccak - used in L1/L2 + computing note hash)
5. authwitness flow
6. L1<>L2 communication,
7. hashing algorithm (Pederson, keccak - used in L1/L2 + computing note hash)
8. authwitness flow
9. Have to trust each application and the recovery contract that their recovery flow is not buggy.
10. Operator needs to know user's decryption key in advance for recovering the private state.

Pros

1. Doesn't require any activity on the user's side!
2. Extensible to any kind of state (e.g. custom notes, public state)

Optional Supplementary reading

Lasse's mass portal migrations