ETHLockbox redesign - Unified report

Summary

Project Name	Optimism Lockbox Redesign
Repository	https://github.com/defi-wonderland/optimism
Commit	9df1fc15

Findings Table

Findings

Aa Severity	Summary
[L-0]	Safety checks are missing to ensure migrations are done correctly
[L-1]	Incorrect Balance Emitted in `LiquidityMigrated` Event
[L-2]	Ether stuck in `OptimismPortal` instead of on `ETHLockbox` if the finalized withdrawal fails
[I-0]	SafeSend contract can receive unexpected Ether before deployment, leading to higher transferred amounts
[I-1]	Missing check at `ETHLockbox#migrateLiquidity` , could check if destination lockbox is authorized
[I-2]	Deprecated .anchors() still used
[I-3]	Typos/Misc
[I-4]	Add check to avoid `lockETH` being called in the context of a withdrawal.

Findings

[L-0] Safety checks are missing to ensure migrations are done correctly

Description

There are certain checks that can be added to reduce the range of errors that can occur during migrations:

1. `OptimismPortal2#migrateToSuperRoots` can add safety checks to ensure the migration is done correctly:
    a. If the lockbox is different than the previous one, it should check the `OptimismPortal2` is already part of the `authorizedPortals` mapping of the lockbox, else deposits and withdrawals may fail until it's authorized. Authorization also checks `superchainConfig` and `proxyOwner` by default. The latter is another missing check, it currently does not check the new lockbox has the same `proxyOwner`, but the `authorizedPortals` would infer it.
    b. If the lockbox is different than the previous one, then it should check the balance of the old lockbox is `0` to ensure it has actually performed the migration. This will require the `ETHLockbox.migrateLiquidity` to be called in the atomic transaction before the call to `OptimismPortal2#migrateToSuperRoots`.
2. `ETHLockbox#authorizeLockbox`:
    a. This function doesn't check that the authorized lockbox has the same `superchainConfig`. This check is present in `ETHLockbox#authorizePortal`. This can lead to a situation where Lockbox B authorizes Lockbox A despite having a different `superchainConfig`, and after the migration, the `OptimismPortal2` that migrated to Lockbox B won't be able to be authorized by Lockbox B as it requires the same `superchainConfig`. The path to roll this back or to become a standard chain is not clear.
        i. The migration paths seem to be oriented around standard chains with the same `superchainConfig`, so I'm assuming the concept of merging lockboxes with different `superchainConfig` is not something that wants to be supported in this iteration. If it were to be, then how to reconciliate these differing chains is not clear. One would have to give ownership to the other, which implies giving control of a lot of funds, reinitialize a multitude of contracts to update the `superchainConfig` so that the portal

can be authorized by the new lockbox, which would make the `initVersion` differ, and lastly have the new lockbox authorize it. All the while, the deposits and withdrawals for the portal would not work.

3. `ETHLockbox#migrateLiquidity`:
    a. This one is trickier and perhaps too tricky to implement for the benefit it gives compared to the others, so I will add it just as food for thought.

       This function is meant to be called atomically with `OptimismPortal2.migrateToSuperRoots`. However, there's no check to ensure the portal is actually pointing at the new lockbox, which can lead deposits going to the previous lockbox, and withdrawals reverting due to lack of liquidity. For a lockbox that only manages a single portal, then this is not very error prone. But if a future can exist where two lockboxes that handle many portals want to merge, ensuring a proper migration won't be as simple.

       It's tricky to implement because it would mean tracking all portals pointing to the lockbox, which is not really trivial, and then looping through all of them to ensure they are pointing at the correct lockbox. In extreme cases this may even end up reaching the block gas limit in a single transaction, although this should be fixable by bundling.

        i. Note for this one: if implemented, it conflicts with the check 1b, given you get a bit of a circular dependency.

[L-1] Incorrect Balance Emitted in `LiquidityMigrated` Event

Description

The `LiquidityMigrated` event incorrectly emits a balance of `0` instead of the actual balance migrated. This happens because `address(this).balance` is used on the emitted event after the liquidity has already been transferred.

Impact

The event log does not accurately reflect the amount of liquidity migrated, this could lead to confusion when tracking liquidity movements on off-chain tools that may misinterpret the migration status.

Recommendation

Store the balance before transferring the liquidity and emit the correct value.

```
function migrateLiquidity(IETHLockbox _lockbox) external {
    ...

    // Store the balance before transferring liquidity.
    uint256 balanceBefore = address(this).balance;

    // Receive the liquidity.
    IETHLockbox(_lockbox).receiveLiquidity{ value: balanceBefore }();

    // Emit the event with the correct balance.
    emit LiquidityMigrated(_lockbox, balanceBefore);
}
```

[L-2] Ether stuck in `OptimismPortal` instead of on `ETHLockbox` if the finalized withdrawal fails

Description

If a finalized withdrawal with a `tx.value > 0` fails on the target call, the Ether will keep stuck in the `OptimismPortal` and won't return back to the `ETHLockbox`.

```
    // Unlock the ETH from the ETHLockbox.
    if (_tx.value > 0) ethLockbox.unlockETH(_tx.value);

    // Trigger the call to the target contract. We use a custom low level method
    // SafeCall.callWithMinGas to ensure two key properties
    //   1. Target contracts cannot force this call to run out of gas by returning a very
```

```
    //     amount of data (and this is OK because we don't care about the returndata
    //   2. The amount of gas provided to the execution context of the target is at lea
    //     gas limit specified by the user. If there is not enough gas in the current cor
    //     to accomplish this, `callWithMinGas` will revert.
    bool success = SafeCall.callWithMinGas(_tx.target, _tx.gasLimit, _tx.value, _tx.d

    // Reset the l2Sender back to the default value.
    l2Sender = Constants.DEFAULT_L2_SENDER;

    // All withdrawals are immediately finalized. Replayability can
    // be achieved through contracts built on top of this contract
    emit WithdrawalFinalized(withdrawalHash, success);

    // Reverting here is useful for determining the exact gas cost to successfully ex
    // sub call to the target contract if the minimum gas limit specified by the user w
    // be sufficient to execute the sub call.
    if (!success && tx.origin == Constants.ESTIMATION_ADDRESS) {
        revert GasEstimation();
    }
}
```

As shown in this code, the Ether amount is pulled from the `ETHLockbox`, then the call is performed. But if the call fails, the Ether is not forwarded to the target and remains and neither sent back to the `ETHLockbox`, remaining on the Portal.

Recommendation

Add the following check after the call

```
if (!success) ethLockbox.lockETH{value: _tx.value}();
```

Impact

It breaks this invariant**.** Nevertheless, can be dismissed as an issue since the funds stuck in the `OptimismPortal` could be sent back to `ETHLockbox` by simply calling `OptimismPortal#migrateLiquidity`, but this doesn't seem to be the expected behavior for the function.

Informationals

[I-0] SafeSend contract can receive unexpected Ether before deployment, leading to higher transferred amounts

When `SafeSend` is deployed by either `SuperchainWETH` or `ETHLiquidity`, Ether can be sent to the contract address to be deployed (which can be pre-calculated). Due to `selfdestruct` behavior, the contract will transfer its entire balance to the recipient, which would be more than the intended `_amount`.

Current Implementation:

```
// SafeSend.sol
contract SafeSend {
    constructor(address payable _recipient) payable {
        selfdestruct(_recipient);
    }
}
```

Proof of Concept:

1. Bob calculates the future address where `SafeSend` will be deployed
2. Bob sends `X` ETH amount to this address
3. Bob calls `SuperchainWETH#relayETH` or `SuperchainWETH#crosschainMint`, where the `SuperchainWETH` or `ETHLiquidity` contracts deploy `SafeSend` with the `_amount` input passed as ETH amount
4. The `selfdestruct` call will send `X` + `_amount` ETH to either the recipient or `SuperchainWETH` address instead of just `_amount`

Impact: Low. While no direct attack vector is identified, this could lead to unexpected behavior where either the recipient or the `SuperchainWETH` contract receives more ETH than intended.

Likelihood: Low. Requires willingness to potentially lose funds without any clear profit.

Recommendation: Add a balance check in the constructor to ensure only the intended amount (`msg.value`) is transferred:

```
// SafeSend.sol
contract SafeSend {
    constructor(address payable _recipient) payable {
        // Send any leftover Ether amount to the address(0)
        uint256 diff = address(this).balance - msg.value;
        if (diff != 0) payable(address(0)).transfer(diff);

        selfdestruct(_recipient);
    }
}
```

[I-1] Missing check at `ETHLockbox.migrateLiquidity`

Description

Currently, only a check exists on the destination lockbox receiving liquidity (`ETHLockbox.receiveLiquidity`) if the source is an authorized lockbox. When migrating liquidity we could also have a check that the destination lockbox is also authorized on the source lockbox.

Impact

Low to none. Acts as an extra security step to make sure both lockboxes are authorized when performing a migration.

Recommendation: Add the following check.

```
if (!authorizedLockboxes[_lockbox]) revert ETHLockbox_Unauthorized();
```

- **[I-2]:** `AnchorStateRegistry#anchors` indicates the function will deprecated and removed in the near future and to use `getAnchorRoot` instead, but the: `OPCM` , `StandardValidator` and `DeployOPChain.s.sol` still use it.

- **[I-3]:** Typos/Misc:
  - `ETHLockbox#LiquidityMigrated` event is missing the `amount` parameter in the natspec
  - `SuperchainConfig` `custom:audit` tag has a typo, should be “contract” instead of contracts.
  - `ETHLockbox#L185#function migrateLiquidity`, `@dev` comment mentions `OptimismPortal.updateLockbox()` which doesn't exist anymore, should be updated to `migrateToSuperRoots`.
  - `OptimismPortal2#L57` , Natspec has a grammar issue, says “If the of this variable”, when it should say “If the value of this variable”.
  - `OptimismPortal2#L372`, missing Natspec for `_newAnchorStateRegistry` param.
- **[I-4]:** Missing check to disallow calling `ETHLockbox#lockETH` on a withdrawal transaction context. This would match the current check in `ETHLockbox#unlockETH`. This would prevent people from misusing this function and be consistent with the `unlockETH` function.