Formal Verification of Vault contract

Summary

This document describes the specification and verification of Vault using the Certora Prover.

The scope of this verification is the Vault.sol (https://github.com/lindy-labs/sc_solidity-contracts/blob/8072f2a60a5a8fef0d106a4d1fdf09788ee522d5/contracts/Vault.sol) contract. Its specification is available here (specs/Vault.spec).

The Certora Prover proved the implementation of the Vault contract is correct with respect to formal specifications written by the security team of Lindy Labs. The team also performed a manual audit of these contracts.

List of Issues Discovered

Severity: Low

Issue:	**Dust in the Vault**
Description:	When user withdraw the principal from the vault, the vault may leave some dust due to rounding of uint division in Solidity. E.g., user withdraws 1000 LUSD, the vault may send only 999.99999... LUSD if the share price is not exactly an integer, which can happen when underlying asset value has changed, e.g., yield earned. A counter example can be found here (https://prover.certora.com/output/52311/0ca7e4fb086d308eeb51?anonymousKey=2a093f38c9406a97d4b9d56e746f07aec63616a8)
Mitigation/Fix:	update the _withdrawSingle internal function to remove the redundant and imprecise computeAmount function call and return the _amount value directly
Property violated:	When user withdraws principal, the requested amount should be sent to the user

Overview of the verification

Description of the Vault contract

The Vault contract is the core of the SandClock system. It allows user to deposit funds to invest and withdraw funds to disinvest. It delegates investing activity to a contract that implements IStrategy (https://github.com/lindy-labs/sc_solidity-contracts/blob/8072f2a60a5a8fef0d106a4d1fdf09788ee522d5/contracts/strategy/IStrategy.sol) interface.

The main value added by Vault is allowing user to specify multiple beneficiaries in a very flexible way, which opens a lot of possibilities. It also allows sponsors to contribute yield to all the users while guaranteeing sponsors to be the last ones to bear investment loss.

It uses shares to keep track of users' entitlements to the underlying assets. Whenever user deposits accepted assets into the vault, the Vault will add a certain amount of shares against the user account. When user withdraws underlying assets, their shares are reduced.

Assumptions and Simplifications

We made the following assumptions during the verification process:

- We unroll loops by max 3 times. Violations that require a loop to execute more than 3 times will not be detected.
- When verifying contracts that make external calls, we assume that those calls can have arbitrary side effects outside of the contracts, but that they do not affect the state of the contract being verified. This means that some reentrancy bugs may not be caught. However, the previous audits should have already covered all the possible reentrancy attacks
- The strategy contract never calls the functions in the Vault.

Verification Conditions

Notation

✔ indicates the rule is formally verified on the latest reviewed commit. Footnotes describe any simplifications or assumptions used while verifying the rules (beyond the general assumptions listed above).

In this document, verification conditions are either shown as logical formulas or Hoare triples of the form {p} C {q}. A verification condition given by a logical formula denotes an invariant that holds if every reachable state satisfies the condition.

Hoare triples of the form {p} C {q} holds if any non-reverting execution of program C that starts in a state satisfying the precondition p ends in a state satisfying the post-condition q. The notation {p} C@withrevert {q} is similar but applies to both reverting and non-reverting executions. Preconditions and post-conditions are similar to the Solidity require and statements.

Formulas relate the results of method calls. In most cases, these methods are getters defined in the contracts, but in some cases they are getters we have added to our harness or definitions provided in the rules file. Undefined variables in the formulas are treated as arbitrary: the rule is checked for every possible value of the variables.

Vault

Rules

1. Integrity of totalShares calculation ✔

```
totalShares == sum(claimers[claimerId].totalShares)
```

2. Integrity of totalPrincipal calculation ✔

```
totalPrincipal == sum(claimers[claimerId].totalPrincipal)
```

3. Claimer shares change only if user deposit, withdraw or claim yield ✔

```
{ s = sharesOf(claimerId) }

  f(e, args);

{
  sharesOf(claimerId) != s =>
    f.selector == deposit
    ||
    f.selector == depositForGroupId
    ||
    f.selector == withdraw
    ||
    f.selector == forceWithdraw
    ||
    f.selector == partialWithdraw
    ||
    f.selector == claimYield
  }
```

4. Claimer principal changes only if user deposit or withdraw ✔

```
{ p = principalOf(claimerId) }

  f(e, args);

{
  principalOf(claimerId) != p =>
    f.selector == deposit
    ||
    f.selector == depositForGroupId
    ||
    f.selector == withdraw
    ||
    f.selector == forceWithdraw
    ||
    f.selector == partialWithdraw
  }
```

5. underlying and getUnderlying() should be the same value ✔

```
underlying() == getUnderlying()
```

6. totalUnderlying() should equal the sum of the Vault and Strategy's balances ✔

```
totalUnderlying() == underlying.balanceOf(currentContract) + underlying.balanceOf(strategy)
```

7. Integrity of deposit function ✔

```
{
    _userBalance = underlying.balanceOf(e.msg.sender);
    _vaultBalance = underlying.balanceOf(currentContract);
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
    _totalSharesOfClaimers = totalSharesOf(claimers);
    _totalPrincipalOfClaimers = totalPrincipalOf(claimers);
}

    deposit(inputToken, lockDuration, amount, pcts, claimers, datas, slippage);

{
    _userBalance - underlying.balanceOf(e.msg.sender) == amount;
    underlying.balanceOf(currentContract) - _vaultBalance == amount;
    totalPrincipal() - _totalPrincipal == amount;
    totalSharesOf(claimers) - _totalPrincipalOfClaimers == amount;
    totalShares() - _totalShares == totalSharesOf(claimers) - _totalPrincipalOfClaimers;
    totalShares() * _totalPrincipal == totalPrincipal() * _totalShares;
}
```

8. Equivalence of deposit function and depositForGroupId function in changing totalShares, totalPrincipal, claimer's shares and principal, and deposits ✔

9. Integrity of withdraw function

```
{
    _userBalance = underlying.balanceOf(to);
    _totalUnderlying = totalUnderlying();
    _totalUnderlyingMinusSponsored = totalUnderlyingMinusSponsored();
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
    _totalDeposits = totalDeposits(depositIds);
    _totalSharesOfClaimers = totalSharesOf(depositIds);
    totalPrincipalOf(depositIds) == _totalDeposits;
}

    withdraw(to, depositIds);

{
    underlying.balanceOf(to) - _userBalance == _totalDeposits;
    _totalUnderlying - totalUnderlying() == _totalDeposits;
    _totalUnderlyingMinusSponsored - totalUnderlyingMinusSponsored() == _totalDeposits;
    _totalPrincipal - totalPrincipal() == _totalDeposits;
    totalDeposits(depositIds) == 0;
    totalPrincipalOf(depositIds) == 0;
    _totalShares - totalShares() == _totalSharesOfClaimers - totalSharesOf(depositIds);
    totalUnderlyingMinusSponsored() == 0 || totalShares() / totalUnderlyingMinusSponsored() ==
_totalShares / _totalUnderlyingMinusSponsored;
}
```

10. Integrity of partialWithdraw function

```
{
    _userBalance = underlying.balanceOf(to);
    _totalUnderlying = totalUnderlying();
    _totalUnderlyingMinusSponsored = totalUnderlyingMinusSponsored();
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
    _totalDeposits = totalDeposits(depositIds);
    _totalSharesOfClaimers = totalSharesOf(depositIds);
    _amount = depositAmount(depositIds[i]);
    totalAmount = totalAmount(amounts);
    totalPrincipalOf(depositIds) == _totalDeposits;
}

    partialWithdraw(to, depositIds, amounts);

{
    underlying.balanceOf(to) - _userBalance == totalAmount;
    _totalDeposits - totalDeposits(depositIds) == totalAmount;
    _totalUnderlying - totalUnderlying() == totalAmount;
    _totalPrincipal - totalPrincipal() == totalAmount;
    _totalShares - totalShares() == _totalSharesOfClaimers - totalSharesOf(depositIds);
    totalPrincipalOf(depositIds) == totalDeposits(depositIds);
    _amount - depositAmount(depositIds[i]) == amounts[i];
    totalUnderlyingMinusSponsored() == 0 || totalShares() / totalUnderlyingMinusSponsored() ==
_totalShares / _totalUnderlyingMinusSponsored;
}
```

11. Integrity of forceWithdraw function ✔

```
{
    _userBalance = underlying.balanceOf(to);
    _totalUnderlying = totalUnderlying();
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
    _totalDeposits = totalDeposits(depositIds);
}

    forceWithdraw(to, depositIds);

{
    underlying.balanceOf(to) >= _userBalance;
    _totalUnderlying >= totalUnderlying();
    _totalPrincipal >= totalPrincipal();
    _totalShares >= totalShares();
    totalDeposits(depositIds) == 0;
}
```

12. Integrity of sponsor function ✔

```
{
    _userBalance = underlying.balanceOf(e.msg.sender);
    _vaultBalance = underlying.balanceOf(currentContract);
    _totalSponsored = totalSponsored();
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
}

    sponsor(inputToken, amount, lockDuration, slippage);

{
    _userBalance - underlying.balanceOf(e.msg.sender) == amount;
    underlying.balanceOf(currentContract) - _vaultBalance == amount;
    totalSponsored() - _totalSponsored == amount;
    _totalShares == totalShares();
    _totalPrincipal == totalPrincipal();
}
```

13. Integrity of sponsor function ✔

```
{
    _userBalance = underlying.balanceOf(e.msg.sender);
    _vaultBalance = underlying.balanceOf(currentContract);
    _totalSponsored = totalSponsored();
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
}

    sponsor(inputToken, amount, lockDuration, slippage);

{
    _userBalance - underlying.balanceOf(e.msg.sender) == amount;
    underlying.balanceOf(currentContract) - _vaultBalance == amount;
    totalSponsored() - _totalSponsored == amount;
    _totalShares == totalShares();
    _totalPrincipal == totalPrincipal();
}
```

14. Integrity of unsponsor function ✔

```
{
    _userBalance = underlying.balanceOf(to);
    _totalUnderlying = totalUnderlying();
    _totalSponsored = totalSponsored();
    _totalDeposits = totalDeposits(depositIds);
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
}

    unsponsor(to, depositIds);

{
    underlying.balanceOf(to) - _userBalance == _totalDeposits;
    _totalUnderlying - totalUnderlying() == _totalDeposits;
    _totalSponsored - totalSponsored() == _totalDeposits;
    _totalShares == totalShares();
    _totalPrincipal == totalPrincipal();
    totalDeposits(depositIds) == 0;
}
```

15. Integrity of partialUnsponsor function ✔

```
{
    _userBalance = underlying.balanceOf(to);
    _totalUnderlying = totalUnderlying();
    _totalSponsored = totalSponsored();
    _deposit = depositAmount(depositIds[i]);
    _totalShares = totalShares();
    _totalPrincipal = totalPrincipal();
}

    partialUnsponsor(to, depositIds, amounts);

{
    underlying.balanceOf(to) - _userBalance == totalAmount(amounts);
    _totalUnderlying - totalUnderlying() == totalAmount(amounts);
    _totalSponsored - totalSponsored() == totalAmount(amounts);
    _deposit - depositAmount(depositIds[i]) == amounts[i];
    _totalShares == totalShares();
    _totalPrincipal == totalPrincipal();
}
```

16. Integrity of privileged settings functions ✔

```
{}

    transferAdminRights(e, newAdmin);
    addPool(token1, pool, tokenI, underlyingI);
    removePool(token2);
    setInvestPct(investPct);
    setTreasury(treasury);
    setPerfFeePct(perfFeePct);
    setStrategy(s);
    setLossTolerancePct(lossTolerancePct);

{
    !hasRole(DEFAULT_ADMIN_ROLE(), e.msg.sender);
    hasRole(DEFAULT_ADMIN_ROLE(), newAdmin);
    getCurvePool(token1) == pool;
    getCurvePool(token2) == 0;
    investPct() == investPct;
    treasury() == treasury;
    perfFeePct() == perfFeePct;
    strategy() == s;
    lossTolerancePct() == lossTolerancePct;
}
```

17. Integrity of withdrawPerformanceFee function ✔

```
{
    _balanceOfTreasury = underlying.balanceOf(treasury());
    _totalUnderlying = totalUnderlying();
    _accumulatedPerformanceFee = accumulatedPerfFee();
}

    withdrawPerformanceFee();

{
    underlying.balanceOf(treasury()) - _balanceOfTreasury == _accumulatedPerformanceFee;
    _totalUnderlying - totalUnderlying() == _accumulatedPerformanceFee;
    accumulatedPerformanceFee() == 0;
}
```

18. Integrity of pause function ✔

```
{}

    pause();

{ paused() }
```

19. Integrity of unpause function ✔

```
{}

    unpause();

{ !paused() }
```

20. Integrity of exitPause function ✔

```
{}

    exitPause();

{ exitPaused() }
```

21. Integrity of exitUnpause function ✔

```
{}

    exitUnpause();

{ !exitPaused() }
```

22. Integrity of yieldFor function ✔

```
{}

    claimableYield = yieldFor(user).yield;
    claimableShares = yieldFor(user).shares;
    perfFee = yieldFor(user).perfFee;
    yield = claimableYield + perfFee;

{
    perfFee == pctOf(yield, perfFeePct());
    yield > 0 => claimableShares > 0;
    claimableShares == 0 => yield == 0;
}
```

23. Correct calculation of maxInvestableAmount ✔

```
totalUnderlying().pctOf(investPct) == investState().maxInvestableAmount
```

24. Withdraw function reverts if the investment is in loss ✔

```
{ totalUnderlyingMinusSponsored() < totalPrincipal() }

    withdraw@withrevert(to, ids);

{ lastReverted }
```

25. Integrity of updateInvested function ✔

```
{ }

    updateInvested();

{ alreadyInvested() == maxInvestableAmount() }
```

26. paused Vault does not accepts any deposit ✔

```
{ paused() }

    deposit@withrevert() || depositForGroupId@withrevert() || sponsor@withrevert()

{ lastReverted }
```

27. deposit function reverts on any invalid param ✔

```
{
    inputToken != getUnderlying() && getCurvePool(inputToken) == 0
    ||
    lockDuration < minLockPeriod()
    ||
    lockDuration > MAX_DEPOSIT_LOCK_DURATION()
    ||
    amount == 0
    ||
    anyZero(pcts)
    ||
    anyZero(beneficiaries)
    ||
    !isTotal100Pct(pcts);
}

    deposit@withrevert(inputToken, lockDuration, amount, pcts, beneficiaries, datas, slippage);

{ lastReverted }
```

28. sponsor function reverts on any invalid param ✔

```
{
    inputToken != getUnderlying() && getCurvePool(inputToken) == 0
    ||
    lockDuration < MIN_SPONSOR_LOCK_DURATION()
    ||
    lockDuration > MAX_SPONSOR_LOCK_DURATION()
    ||
    amount == 0;
}

    sponsor@withrevert(inputToken, amount, lockDuration, slippage);

{ lastReverted }
```

29. exitPaused Vault rejects any withdrawals ✔

```
{ exitPaused() }

   withdraw@withrevert()
   ||
   partialWithdraw@withrevert()
   ||
   forceWithdraw@withrevert()
   ||
   claimYield@withrevert()
   ||
   unsponsor@withrevert()
   ||
   partialUnsponsor@withrevert()

{ lastReverted }
```

30. Correct calculation of totalUnderlyingMinusSponsored ✔

```
totalUnderlyingMinusSponsored() == totalUnderlying() - totalSponsored() - accumulatedPerfFee()
```

31. withdraw function reverts if the lock period is not passed yet ✔

```
{
   depositLockedUntil(ids[0]) > e.block.timestamp
   ||
   depositLockedUntil(ids[1]) > e.block.timestamp
   ||
   depositLockedUntil(ids[2]) > e.block.timestamp
}

   withdraw@withrevert(e, to, ids)

{ lastReverted }
```

32. withdraw function reverts if the user didn't make the deposit ✔

```
{
   depositOwner(ids[0]) != e.msg.sender
   ||
   depositOwner(ids[1]) != e.msg.sender
   ||
   depositOwner(ids[2]) != e.msg.sender
}

   withdraw@withrevert(e, to, ids)

{ lastReverted }
```

33. privileged functions should revert if the msg.sender does not have the privilege ✔

```
{
    adminFunctions(f) && !hasRole(DEFAULT_ADMIN_ROLE(), e.msg.sender)
    ||
    settingsFunctions(f) && !hasRole(SETTINGS_ROLE(), e.msg.sender)
    ||
    keeperFunctions(f) && !hasRole(KEEPER_ROLE(), e.msg.sender)
    ||
    sponsorFunctions(f) && !hasRole(SPONSOR_ROLE(), e.msg.sender)
}

    f@withrevert(e, args)

{ lastReverted }
```

34. deposit function reverts if the vault is in a loss ✔

```
{ totalUnderlyingMinusSponsored() < applyLossTolerance(totalPrincipal()) }

    deposit@withrevert(e, args)

{ lastReverted }
```