Formal Verification of LiquityStrategy contract

Summary

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

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

The Certora Prover proved the implementation of the LiquityStrategy 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

None

Overview of the verification

Description of the LiquityStrategy contract

The LiquityStrategy contract generates yield by investing LUSD assets into Liquity Stability Pool contract. Stability pool gives out LQTY & ETH as rewards for liquidity providers. The LQTY rewards are normal yield rewards. ETH rewards are given when liquidating Troves using the LUSD we deposited. When liquidation happens, our balance of LUSD goes down and we get an 1.1x (or higher) value of ETH. In short, we make a 10% profit in ETH every time our LUSD is used for liquidation by the stability pool.

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 satsifying the precondition p ends in a state satisfying the postcondition q. The notation {p} C@withrevert {q} is similar but applies to both reverting and non-reverting executions. Preconditions and postconditions 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.

[*] Rule 4 does not pass the verification because the function has a call function to external contract with arbitrary calldata, and Certora assumes that call can modify other contract's state. Certora cannot summerize the call function with HAVOC_ECF either. In this case it can modify the balances of the underlying ERC20 token.

LiquityStrategy

Rules

1. Privileged functions revert if no 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)
    ||
    managerFunctions(f) && !hasRole(MANAGER_ROLE(), e.msg.sender)
}

    f@withrevert(e, args);

{
    lastReverted
}
```

2. initialize function can be called once only ✔

```
{}

    initialize(_vault, _admin, _stabilityPool, _lqty, _underlying, _keeper, _principalProtectionPct,
_curveExchange);
    initialize@withrevert(_vault, _admin, _stabilityPool, _lqty, _underlying, _keeper, _principalProtectionPct,
_curveExchange);

{ lastReverted }
```

3. invest function moves all the LUSD balance from strategy to the stability pool ✔

```
{
    _balanceOfStrategy = underlying.balanceOf(currentContract);
    _balanceOfPool = underlying.balanceOf(stabilityPool);
}

    invest();

{
    underlying.balanceOf(currentContract) == 0;
    underlying.balanceOf(stabilityPool) - _balanceOfPool == _balanceOfStrategy;
}
```

4. [*] reinvest function converts LQTY and ETH rewards to LUSD and moves all the LUSD balance from strategy to the stability pool

```
{
    _balanceOfStrategy = underlying.balanceOf(currentContract);
    _balanceOfPool = underlying.balanceOf(stabilityPool);
}

    invest();

{
    underlying.balanceOf(currentContract) == 0;
    underlying.balanceOf(stabilityPool) - _balanceOfPool >= _balanceOfStrategy;
}
```

5. withdrawToVault(uint256 amount) should withdraw LUSD to the Vault ✔

```
{
    stabilityPool != currentContract && vault != currentContract && vault != stabilityPool;
    underlying.balanceOf(currentContract) == 0;
    _balanceOfPool = underlying.balanceOf(stabilityPool);
    _balanceOfVault = underlying.balanceOf(vault);
    stabilityPool.getCompoundedLUSDDeposit(currentContract) == _balanceOfPool;
}

    amountWithdrawn = withdrawToVault(amount)

{
    underlying.balanceOf(currentContract) == 0;
    _balanceOfPool - amountWithdrawn == underlying.balanceOf(stabilityPool);
    _balanceOfVault + amountWithdrawn == underlying.balanceOf(vault);
}
```

6. havest function claims ETH and LQTY rewards only and not change LUSD balance of any account ✔

```
{
    stabilityPool != currentContract && vault != currentContract && vault != stabilityPool;
    _balanceOfPool = underlying.balanceOf(stabilityPool);
    _balanceOfStrategy = underlying.balanceOf(currentContract);
    _balanceOfVault = underlying.balanceOf(vault);
    _lqtyBalance = lqty.balanceOf(currentContract);
    _ethBalance = getEthBalance();
}

    harvest();

{
    _balanceOfPool == underlying.balanceOf(stabilityPool);
    _balanceOfStrategy == underlying.balanceOf(currentContract);
    _balanceOfVault == underlying.balanceOf(vault);
    lqty.balanceOf(currentContract) >= _lqtyBalance;
    getEthBalance() >= _ethBalance;
}
```

7. investedAssets is greater than or equal to the LUSD deposit in the stability pool ✔

```
investedAssets() >= stabilityPool.getCompoundedLUSDDeposit(currentContract)
```

8. hasAssets return value should be consistent with investedAssets return value ✔

```
investedAssets() > 0 <=> hasAssets() && investedAssets() == 0 <=> !hasAssets()
```

9. the strategy is sync ✔

```
isSync()
```

10. allowSwapTarget(address _swapTarget) should whitelist the _swapTarget ✔

```
{}

    allowSwapTarget(_swapTarget)

{ allowedSwapTargets(_swapTarget) == true }
```

11. denySwapTarget(address _swapTarget) should remove the _swapTarget from the whitelist ✔

```
{}

    denySwapTarget(_swapTarget)

{ allowedSwapTargets(_swapTarget) == false }
```

12. setMinProtectedAssetsPct should set minProtectedAssetsPct ✔

```
{}

    setMinProtectedAssetsPct(pct)

{ minProtectedAssetsPct() == pct }
```

13. transferAdminRights should transfer admin roles from msg sender to the new admin ✔

```
{}

    transferAdminRights(newAdmin)

{
    !hasRole(DEFAULT_ADMIN_ROLE(), e.msg.sender);
    !hasRole(KEEPER_ROLE(), e.msg.sender);
    !hasRole(SETTINGS_ROLE(), e.msg.sender);
    hasRole(KEEPER_ROLE(), newAdmin);
    hasRole(DEFAULT_ADMIN_ROLE(), newAdmin);
    hasRole(SETTINGS_ROLE(), newAdmin);
}
```

14. invest reverts if the underlying asset balance is 0 ✔

```
{ underlying.balanceOf(currentContract) == 0 }

    invest()

{ lastReverted }
```

15. withdrawToVault reverts if the amount is 0 or greater than investedAsset ✔

```
{ amount == 0 || amount > investedAssets() }

    withdrawToVault(amount)

{ lastReverted }
```