Formal Verification of RyskStrategy contract

Summary

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

The scope of this verification is the RyskStrategy.sol (https://github.com/lindy-labs/sc_solidity-contracts/blob/abe1005872ea3d49052d4ede3b643fb14d4f8542/contracts/strategy/rysk/RyskStrategy.sol) contract. Its specification is available here (specs/RyskStrategy.spec).

The Certora Prover proved the implementation of the RyskStrategy 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 RyskStrategy contract

RyskStrategy generates yield by investing into a Rysk LiquidityPool, that serves to provide liquidity for a dynamic hedging options AMM.

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.

RyskStrategy

Rules

1. Privileged functions should revert if there is no privilege ✔

```
{
    adminFunctions(f) && !hasRole(DEFAULT_ADMIN_ROLE(), e.msg.sender)
    ||
    keeperFunctions(f) && !hasRole(DEFAULT_ADMIN_ROLE(), e.msg.sender)
    ||
    managerFunctions(f) && !hasRole(MANAGER_ROLE(), e.msg.sender);
}
    f@withrevert(e, args);
{
    lastReverted;
}
```

2. When the balance of the current contract is zero, invest should revert ✔

```
{
    require underlying.balanceOf(currentContract) == 0;
}
    invest@withrevert(e);
{
    lastReverted;
}
```

3. Invest function should perform a deposit in the Rysk Liquidity Pool ✔

```
{
    ryskLqPool != currentContract;
}
    _balanceOfStrategy = underlying.balanceOf(currentContract);
    _balanceOfPool = underlying.balanceOf(ryskLqPool);
    invest(e);
{
    underlying.balanceOf(currentContract) == 0;
    underlying.balanceOf(ryskLqPool) - _balanceOfStrategy == _balanceOfPool;
}
```

4. When the amount is zero, withdrawToVault should revert ✔

```
{
    require amount == 0;
}

    withdrawToVault@withrevert(e, amount);

{
    lastReverted;
}
```

5. Initiate a withdraw to the Rysk Liquidity Pool of the number of underlying ERC20 ✔

```
{
    require ryskLqPool != currentContract && vault != currentContract && vault != ryskLqPool;

    _balanceOfRyskLqPool = underlying.balanceOf(ryskLqPool);
    _balanceOfVault = underlying.balanceOf(vault);
    _balanceOfStrategy = underlying.balanceOf(currentContract);
}

    withdrawToVault(e, amount);

{
    _balanceOfStrategy == underlying.balanceOf(currentContract);
    underlying.balanceOf(vault) - _balanceOfVault == _balanceOfRyskLqPool -
underlying.balanceOf(ryskLqPool);
}
```

6. Complete a withdraw initiated in a previous epoch ✔

```
{
    require vault != currentContract && vault != ryskLqPool && currentContract != ryskLqPool;
    _balanceOfVault = underlying.balanceOf(vault);
    _balanceOfRyskLqPool = underlying.balanceOf(ryskLqPool);
    _balanceOfStrategy = underlying.balanceOf(currentContract);
}

    completeWithdrawal(e);

{
    _balanceOfStrategy == underlying.balanceOf(currentContract);
    _balanceOfRyskLqPool >= underlying.balanceOf(ryskLqPool);
    underlying.balanceOf(vault) >= _balanceOfVault;
}
```

7. isSync() should return false ✔

```
!isSync();
```

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

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

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

```
{
}

    transferAdminRights(e, newAdmin);

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