

Formal Verification Report of Lyra

Summary

This document describes the specification and verification of **Lyra** using Certora Prover. The work was undertaken from **12, May 2021 - 2, June 2021**.

The scope of our verification was the Option Market and the Liquidity Pool contracts. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations.

List of Main Issues Discovered

Severity: Critical

Issue:	Significant loss of system assets
Description:	In the case of withdrawal between rounds, one can withdraw the same liquidity certificate for unlimited number of times, thus draining the system. Furthermore, that liquidity certificate does not have to be under our ownership - anybody can withdraw a certificate even if they do not own it.
Mitigation:	The used liquidity certificate is now being burned, and the sender's ownership of that liquidity certificate is being verified

Severity: High

Issue:	Denial of service of the system
Rule:	Burnable tokens must not exceed total supply
Description:	A user can mark multiple times his liquidity certificate for withdrawal, causing wrong interpretation of assets to free. This can cause the round end process to revert, leaving the system in a denial of service state.
Mitigation:	<code>signalWithdrawal()</code> now reverts when it is given a liquidity certificate that has already signaled exit.

Severity: Low

Issue:	Unsafe casts from <code>uint</code> to <code>int</code>
Description:	The functions <code>openPosition()</code> and <code>closePosition()</code> perform unsafe casts of amount from <code>uint</code> to <code>int</code> . An overflow can lead to unexpected behavior, including wrong changes to the contract internal state.
Mitigation:	The requirement <code>int(amount) > 0</code> was added to <code>_canDoTrade()</code> .

Severity: Low

Issue:	Wrong revert reason
Description:	<p>In <code>liquidateExpiredBoard()</code>, the for loop that removes the board from the list of live boards is <code>for (uint i = liveBoards.length - 1; i >= 0; i--)</code> and since <code>i</code> is of type <code>uint</code>, the condition <code>i >= 0</code> will always pass. Then if the board is not in the list of live boards, the <code>i--</code> operation after the last loop iteration will cause a revert due to underflow, instead of reverting in the later <code>require(popped)</code> statement.</p> <p>A similar issue occurs when the system is between rounds. In this case, there will also be a revert due to underflow, this time in the loop initialization line <code>uint i = liveBoards.length - 1</code>, because the list of live boards would be empty.</p>
Mitigation:	The for loop was modified to <code>for (uint i = 0; i < liveBoards.length; i++)</code> .

Severity: Low

Issue:	Ignoring return value from <code>Synthetic.exchange</code>
Description:	Does not handle the case when the call fails
Mitigation:	Fixed

Severity: Recommendation


Issue:	"Buying" options for zero cost
Description:	Due to additional fees the system may evaluate certain options to Zero value, and is willing to buy those options for Zero cost. We recommend refraining from Zero cost trades.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, but provide no warranty of any kind, express or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, or otherwise, arising from, out of or in connection with the results reported here.

Notations

1.
 - a. indicates the rule is formally verified on the latest commit.
 - b. We write * when the rule was verified on a simplified version of the code (or under some assumptions).
 - c. * indicates that the rule was violated at some version of the code.
2.  indicates the rule was violated under the current tested version of the code.
3. indicates the rule is not yet formally specified.
4. indicates the rule is postponed (<due to other issue, low priority?>).
5. We use [Hoare triples](#) of the form $\{p\} C \{q\}$, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.

The syntax $\{p\} (C_1 \ C_2) \{q\}$ is a generalization of Hoare rules, called [relational properties](#). $\{p\}$ is a requirement on the states before C_1 and C_2 , and $\{q\}$ describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \text{op} C_2$, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of LiquidityPool and OptionMarket

Liquidity Pool

When an LP supplies collateral to the pool, those funds are used to sell options against, as well as for hedging delta risk. The pool needs certainty that until all options that were sold against that collateral have expired, the collateral will remain in the pool. When an LP wishes to withdraw, they must signal their intention to withdraw, and their funds will become available for withdrawal at the moment that the latest-expiring option expires.

When an LP enters the pool, they are issued a liquidity certificate (an ERC721 NFT) which represents a share in the pool, and which, when the LP exits, will be burnt in exchange for their share of the collateral in the pool.

Liquidity pool handles the start and the end of the rounds. The conclusion of a round comes about when all boards are liquidated. As part of the liquidation process; all options are cash settled, with all capital required for payouts or refunded short collateral being reserved for users.

Option Market

A collection of boards and listings which a user can trade against. Manages collateral locking/freeing for longs, holds collateral for shorts.

Also, manages board expiry and liquidation, cash settlement, and option exercise that finishes in the money.

Traders have both the ability to buy a call/put option from the market or sell them to the market. When options are purchased from the market, collateral is locked in the `LiquidityPool`, and the user sends the cost to the `LiquidityPool`. When options are sold to the market, the user must lock collateral in the `OptionMarket` (this contract).

Beyond trading, the `OptionMarket` is also responsible for the liquidation of assets held by a board as well as reserving the correct amount of assets required to pay out to all the traders. This involves holding onto the difference of `spotPrice` and `strike` for long options and liquidating and sending a portion of the collateral from options sold to the market.

Properties

The following properties were formally defined.

1. **Tokens to burns at end of round cannot be more than the total supply** ^x

The total amount of tokens signal to withdraw is less than the total tokens supplied.

```
tokensBurnableForRound()  totalTokenSupply()
```

2. **Change to only one** `certificateID`

Each operation changes at most one certification.

```
{ certificateId1  certificateId2
  cId1Before = getCertificateIdLiquidity(certificateId1)
  cId2Before = getCertificateIdLiquidity(certificateId2) }
op
{ ¬ (cId1Before  getCertificateIdLiquidity(certificateId1)
    cId2Before  getCertificateIdLiquidity(certificateId2) )
}
```

3. **Either withdraw or signal withdraw, can't do both**

At any given state, a certification id can be either signaled to withdraw or withdrew, not both.

```
{ block.timestamp > 0 }
  r1 = signalWithdrawal(certificateId) ~  r2 = withdraw(b,
certificateId)
{ !(r1  r2) }
```

4. **Validity of rounds**

a. **Creating a board results in a valid board id**

```
{ }
  boardId = createOptionBoard(e, args)
{ boardIdExists(boardId) }
```

b. **Opening a new round results in non zero live boards**

```
{ getLiveBoardsLength() = 0 }
  createOptionBoard(e, args);
{ getLiveBoardsLength() > 0 }
```

c. While in a round all boards created are to be closed by the end of the round

```
{ }
  boardId = createOptionBoard(e, args)
{ getOptionBoardExpiry(boardId)  getMaxExpiryTimestamp() }
```

d. The system can reach the in-between rounds state

```
{ getLiveBoardsLength() = 1 }
  liquidateExpiredBoard(e, boardId);
{ getLiveBoardsLength() = 0 }
```

e. Integrity of the end of a round

```
getLiveBoardsLength() = 0  getBlockTimestamp(e) >
getMaxExpiryTimestamp()
```

5. Validity of Live boards

a. Integrity of live boards representation

```
( index < getLiveBoardsLength()      boardId = getLiveBoard
(index) )
  ( getOptionBoardId(boardId) == boardId
  getOptionBoardExpiry(boardId) > 0 )
```

b. Uniqueness of board id

```
getLiveBoard(i) = getLiveBoard(j)  i = j
```

c. No empty boards

```
getLiveBoardsLength() > i  getListingLength(i) > 0
```

6. Validity of listing

```
getOptionBoardListingId(boardId, i) = listingId  
getOptionListingBoardId(listingId) = boardId
```

7. Can't open position with listing id who's board was liquidated
8. Once opening a position, one can close the position
9. One can not open an opposite position to an existing one
10. Validity of pricing
 - a. Increase in implied volatility necessarily increase the option price
 - b. Decrease in time to expiration necessarily decrease the option price