Lyra
Smart Contract Audit

Lyra

07 July 2021

iosiro

1. Introduction

iosiro was commissioned by Lyra to conduct a smart contract audit on the Lyra smart contracts. The audit was performed by two auditors between 17 and 31 May 2021, consuming a total of 20 resource days.

This report is organized into the following sections.

- **Section 2 - Executive summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit details:** A description of the scope and methodology of the audit.
- **Section 4 - Design specification:** An outline of the intended functionality of the smart contracts.
- **Section 5 - Detailed findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to better understand the risk exposure of the smart contracts, and as a guide to improving the security posture of the smart contracts by remediating issues identified. The results of this audit are only a reflection of the source code reviewed at the time of the audit and of the source code that was determined to be in-scope.

The purpose of this audit was to achieve the following:

- Identify potential security flaws.
- Ensure that the smart contracts functioned according to the documentation provided.

Assessing the off-chain functionality associated with the contracts, for example, backend web application code, was out of scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered very high risk with regards to cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed where possible.

2. Executive summary

This report presents the findings of an audit performed by iosiro on the smart contract implementation of Lyra. Lyra is a layer 2 options platform allowing users to trade various options based on synthetic assets.

Audit findings

Overall, the system was found to operate as intended and the implementation was of a high standard.

- Two medium risk issues relating to a function revert and a withdrawal issue were found and remediated during the audit.
- Several low and informational items relating to best practices were risk-accepted or remediated.

There was some dependency on the administrator role within the Lyra system, with users relying on the administrators to create new boards and set other global parameters related to option pricing. Additionally, it was understood that an external bot will be deployed to ensure that liquidity providers remain delta neutral, exchange assets, and that the boards are kept up to date. However, these functions were not access controlled and could be invoked by any user, allowing the system to operate correctly without the bot.

Recommendations

At a high level, the security posture of the Lyra system could be further strengthened by:

- Testing the system integration once Synthetix functionality for shorting is available on L2.
- Performing additional audits at regular intervals, as security best practices, tools, and knowledge change over time. Additional audits over the course of the project's lifespan ensure the longevity of the codebase.
- Creating a bug bounty program to encourage the responsible disclosure of security vulnerabilities in the system.

3. Audit details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files was considered to be out-of-scope. Out-of-scope code that interacts with in-scope code was assumed to function as intended and not introduce any functional or security vulnerabilities.

3.1.1 Smart contracts

- **Project name:** Lyra
- **Commit:** 80c739c98ce02982a2214f22e477aa9dd5434d27
- **Final Review Commit:** 0a64a5df9a806b8c2710c3eb7badcae6d8591dc6
- **Files:** LiquidityCertificate.sol, LiquidityPool.sol, LyraGlobals.sol, OptionGreekCache.sol, OptionMarket.sol, OptionMarketPricer.sol, PoolHedger.sol, ShortCollateral.sol

3.2 Methodology

A variety of techniques, described below, were used to conduct the audit.

3.2.1 Code review

The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high risk areas of the system.

3.2.2 Dynamic analysis

The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided. Manual analysis was used to confirm that the code was functional and to discover whether any potential security issues identified could be exploited. The coverage report of the provided Hardhat tests as on the final day of the audit is given below.

File	**% Stmts**	**% Branch**	**% Funcs**	**% Lines**	**Uncovered Lines**
BlackScholes.sol	100	100	100	100	
LiquidityCertificate.sol	100	95	100	100	
LiquidityPool.sol	95.18	71.88	87.5	93.41	... 631,635,636
LyraGlobals.sol	100	100	100	100	
OptionGreekCache.sol	98.59	81.82	100	98.63	477,502
OptionMarket.sol	100	97.3	100	100	

OptionMarketPricer.sol	100	85.71	100	100	
PoolHedger.sol	89.62	75	88.24	89.62	... 201,202,312
ShortCollateral.sol	90	64.71	100	89.74	62,71,80,89

Overall, the code coverage of the unit and integration tests were expansive. Only some defensive programming statements and utility view functions were not covered.

3.2.3 Automated analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. The static analysis results were manually reviewed and any false positives were removed from the results. Any true positive results were included in this report.

Static analysis tools commonly used include Slither, Securify, and MythX. Tools such as the Remix IDE, compilation output, and linters could also be used to identify potential areas of concern.

3.3 Risk ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High risk** - The issue could result in a loss of funds for the contract owner or system users.

- **Medium risk** - The issue resulted in the code specification being implemented incorrectly.
- **Low risk** - A best practice or design issue that could affect the security of the contract.
- **Informational** - A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** - The issue was identified during the audit and has since been addressed to a satisfactory level to remove the risk that it posed.

4. Design specification

The following section outlines the intended functionality of the system at a high level.

Optimism OVM

The Lyra system was designed and developed to be deployed to the Optimism OVM and integrate with Synthetix L2 to exchange assets and open short positions.

At the time of the review, changes to the Synthetix code base which would enable shorting on the Optimism mainnet (SIP-135) were not yet complete. For this reason, a best-effort approach was taken to assess the integration between Lyra and Synthetix Shorts.

A pivotal difference between Sythentix on Ethereum mainnet and Optimism is that there is no settlement period for exchanges on Optimism. Consequently, the Lyra code was regarded as incompatible with Ethereum mainnet and only the behavior of Synthetix L2 was considered during the audit.

Option Markets

The Option Market is a collection of listings which a user can trade against. The Option Market consists of rounds, with each round starting with the creation of a new board when there are no active boards and ending when the last board is liquidated. Only the owner of the Option Market can publish new boards (`OptionMarket.createOptionBoard(...)` and it is assumed that the Lyra team will adequately initialize the boards to ensure fair pricing. The maximum length of a round is 10 weeks. A round is completed when all the boards in the Option Market have expired and been liquidated (`OptionMarket.liquidateExpiredBoard(...)`).

A board can be liquidated at any time, by anyone, after its expiry date, which locks in the spot price for the board's expired options. As such, the system relies on users and bots liquidating boards in a timely manner. Similarly, users can only open or close options if the pricing information is not stale, which requires users, bots or the Lyra team to regularly call `OptionGreekCache.updateAllStaleBoards()`.

The Option Markets supports four different types of options:

- Long Call
- Short Call
- Long Put
- Short Put

Long calls require the market to be fully collateralized, and as such, underlying liquidity is used to purchase the assets from Synthetix based on the amount of options bought. Similarly, long puts require the underlying quote asset to be collateralized from the liquidity pool. Short calls and short put premiums are sent to the user from the liquidity pool, while the user provides the assets to be locked up for the duration of the option. Options are cash settled as opposed to physical delivery, meaning that only the difference between the strike price and the spot price locked at expiry is sent back to the user.

The prices for long options and the premiums for short options are dependent on the initial parameters as set by the board and the underlying asset price, with the exact pricing determined by the BlackScholes model. Up until expiry, users have the ability to close some or all of their open options, with the price calculated by the BlackScholes contract to determine a fair price.

Liquidity Pool and Certificates

Liquidity for the Option Market is provided by users depositing the quote asset, usually `sUSD`, into the liquidity pool. For each deposit, the user is issued a `LiquidityCertificate` containing the amount and deposit date of the liquidity. These certificates are implemented as Non-Fungible Tokens (NFTs) and conform to the ERC-721 token standard.

Users can withdraw their liquidity between rounds by invoking the `LiquidityPool.withdraw(...)` function. If a user wishes to withdraw at a later point, they can signal their intent using `LiquidityPool.signalWithdrawal(...)`, which will exclude their liquidity from future rounds. This action can be undone by calling `LiquidityPool.unSignalWithdrawal(...)`. Partial withdrawals are not supported; instead, `LiquidityCertificates` can be split (`LiquidityCertificate.split(...)`) and individually withdrawn.

The `LiquidityPool` contract makes use of a token price and supply concept to calculate and keep track of the effective value of each liquidity certificate. This token supply is separate to the total number of liquidity certificates minted, as each certificate represents a different amount of liquidity. Instead, the value of the certificate at withdrawal is calculated using the token price when the certificate was minted and the last round that the certificate was included in the liquidity pool.

Hedging

When initialized, the system will make use of Synthetix shorts to minimize the risk exposure of the underlying asset composition to liquidity providers. The amount of assets to hedge is calculated using the global `netDelta` of the Options Market. Any user can invoke the `PoolHedger.hedgeDelta()` function, but it is expected that the Lyra team will regularly invoke this function. At the end of a round, the `netDelta` always returns to 0 and the short position is set to 0, which includes the withdrawal of any collateral from Synthetix.

Governance

The only privileged role in the system is the owner, who can add boards and update configuration values in the global state.

The owner can also pause the system, which prevents users from opening or closing positions, liquidating boards, or updating the current hedge. Pausing the system does not prevent liquidity providers from withdrawing their available liquidity or users from exercising expired options.

Upgradeability

The Lyra smart contracts do not make use of an upgradeable proxy pattern and there are no functions to transfer state from an existing deployment to a new one. When new versions of the system are deployed, liquidity providers will need to withdraw their liquidity and redeposit into the new system.

5. Detailed findings

The following section details the findings of the audit.

5.1 High risk

No high risk issues were present at the conclusion of the review.

5.2 Medium risk

No medium risk issues were present at the conclusion of the review.

5.3 Low risk

No low risk issues were present at the conclusion of the review.

5.4 Informational

5.4.1 Minute amounts of assets remain in `ShortCollateral` due to precision loss

ShortCollateral.sol#L117, *ShortCollateral.sol#L130*

Description

After a board had been liquidated and all users had exercised their options, small amounts of both quote and base assets were left in the `ShortCollateral` contract. This was due to precision loss when calculating the amounts that needed to be transferred.

As an example, the `ShortCollateral` contract's balances *in base units* at the end of a 10-round test scenario were found to be:

```
Address: 0x8A791620dd6260079BF849Dc5567aDC3F2FdC318, 3 sUSD, 2
sETH
```

Analysis revealed this resulted from values being rounded down when exercising the options.

Recommendation

An `onlyOwner` function to calculate and transfer the difference between the total outstanding option balances and the contract's balances could be introduced. However, it should be determined if such functionality is worth the overhead costs or whether the minute amounts are acceptable losses to the liquidity providers.

Update

This issue was risk-accepted, as the minute remaining amounts ensure that the user's withdrawals are not blocked.

5.4.2 Quote asset remaining in `LiquidityPool` after complete withdrawal

LiquidityPool.sol#L265-L279

Description

Minimal amounts of quote assets would remain in the `LiquidityPool` contract after all liquidity providers have burned their liquidity certificates and all options have been exercised. Analysis showed that this discrepancy was due to precision loss in the token price.

The code responsible for this is shown below:

```
uint currentRoundValue = expiryToTokenValue[maxExpiryTimestamp];
uint value;
// If they haven't signaled withdrawal, and it is between rounds
if (certificateData.burnableAt == 0 &amp;&amp; currentRoundValue != 0) {
    uint tokenAmt = certificateData.liquidity.divideDecimal(enterValue);
    totalTokenSupply = totalTokenSupply.sub(tokenAmt);
    value = tokenAmt.multiplyDecimal(currentRoundValue);
    liquidityCertificate.burn(msg.sender, certificateId);
    require(quoteAsset.transfer(beneficiary, value));
    emit Withdraw(beneficiary, certificateId, value, totalQuoteAmountReserved);
    return value;
}
```

The price per token, retrieved and stored in `currentRoundValue` was calculated at the end of each round. Any loss in precision was then amplified when multiplying the value by the token amount at `LiquidityPool.sol#L274`. Fortunately, the total amount of assets that might remain was bound to value of the `totalTokenSupply`, which for most Synthetix assets would be an insignificant amount.

Recommendation

If the loss is deemed significant enough to address, a statement should be added to transfer the remaining balance of the contract when the `totalTokenSupply` is zero and the total pool liquidity is equal to current balance (No outstanding `totalQuoteAmountReserved` or `queuedQuoteFunds`).

Update

This issue was risk-accepted to prevent locking users from withdrawing their liquidity.

5.4.3 `LiquidityCertificates.certificates(...)` function vulnerable to denial-of-service

LiquidityCertificate.sol#L59

Description

The `LiquidityCertificates.certificates(...)` function returned all of the liquidity certificates belonging to a given user. However, it was possible to issue these certificates to another user with a minimum of 1 sUSD, making it inexpensive to send several low-liquidity certificates to another user. With a sufficient number of certificates, the function would revert due to hitting the transaction gas limit.

Recommendation

The `minLiquidity` should be increased, making it expensive for a user to attempt to prevent others from viewing their certificates.

Update

The team marked this issue as accepted, as users are also able to list their certificates by checking the events emitted when liquidity was added.

5.4.4 Front-runnable initializers

LiquidityPool.sol#L127, *OptionGreekCache.sol#L115*, *OptionMarket.sol#L208*, *OptionMarketPricer.sol#L61*, *PoolHedger.sol#L82*, *ShortCollateral.sol#L51*, *OptionMarketViewer.sol#L66*

Description

All contract initializers were missing access controls, allowing any user to initialize the contract. By front-running the contract deployers to initialize the contract, the incorrect parameters may be supplied, leaving the contract needing to be redeployed.

Recommendation

While the code that can be run in OVM constructors is limited, setting the `owner` in the contract's constructor to the `msg.sender` and adding the `onlyOwner` modifier to all initializers would be a sufficient level of access control.

Update

The team indicated that this issue is an accepted risk due to OVM contract size constraints.

5.4.5 Delayed asset exchanges could expose liquidity providers to additional risk

LiquidityPool.sol#L323)

Description

After the initial audit, changes were made to the manner in which assets were exchanged within the `LiquidityPool`. Assets are no longer automatically exchanged when a position is opened or closed, instead a new function `LiquidityPool.exchangeBase()` was introduced, due to the gas constraints of the OVM. This function does not have any access control.

Since assets are no longer automatically exchanged, liquidity providers are more exposed to changes in the base asset price. For this reason it is crucial that function is invoked regularly, either by users or the Lyra bot.

Recommendation

Adding a staleness check to the `LiquidityPool.lockBase(...)` and `LiquidityPool.freeBase(...)` functions that reverts if the time since last exchange exceeds some threshold would prevent opening or closing of positions until `LiquidityPool.exchangeBase()` is executed again. This provides a mechanism to encourage users to invoke `LiquidityPool.exchangeBase()` in the event that the bot is unable to.

Update

The team indicated that this issue is an accepted risk, as it will be mitigated by the fact that the dApp front-end will invoke `LiquidityPool.exchangeBase()` after any Long Calls are opened or closed and that the bot will be configured to invoke the function every minute if necessary.

5.4.6 Design comments

Actions to improve the functionality and readability of the codebase are outlined below.

`ReentrancyGuard`

It is recommended that any external function implement OpenZeppelin's `ReentrancyGuard` to protect against reentrancy attacks. While no issues were identified due to the use of the checks-effects pattern, it is recommended that the modifier still be used as a defense-in-depth measure.

Update

Due to contract size limitations for OVM deployment, this risk was accepted.

Missing validation

The following function did not validate if the passed parameters were not the zero address:

1. OptionMarketPricer.init() did not validate that the `_optionMarket` address is not zero.

Update

This issue was accepted, as any contracts with incorrect initializer calls will be redeployed.

Missing require messages

Several `require` statements across the codebase were missing revert messages. These messages help debug transactions that do not meet the requirement, and should be added where required.

Update

Due to contract size limitations for OVM deployment, this risk was accepted.

5.5 Closed

5.5.1 `SetShortTo(...)` will revert when collateral is zero (medium risk)

PoolHedger.sol#L291

Description

A discrepancy between the `TestShortCollateral` and Synthetix's `CollateralShort` contracts was identified that would have resulted in the `SetShortTo` function reverting when deployed to mainnet. Synthetix required that a loan's collateralization ratio (C-Ratio) is above a certain threshold, otherwise risking liquidation. The condition prevented loans from having 0 collateral even when the drawn amount was also 0.

Recommendation

Add a statement to require that the collateral is never set to zero, by incrementing the desired collateral by 1 when it is 0.

Update

The Lyra team confirmed the issue and addressed it by collaborating with the Synthetix team to remove the requirement that the C-Ratio has to be met when the loan amount is 0. This change was included in e1650e0 and merged by the Synthetix team. The update should be included in the initial deployment of Synthetix on the Optimism mainnet.

5.5.2 Return statement prevents complete withdrawal of liquidity (medium risk)

PoolHedger.sol#L163

Description

A return statement in `PoolHedger.updatePosition` resulted in the function returning prematurely and not transferring the remaining quote asset balance to the liquidity pool. It also skipped the code that emits an event when the net delta is updated. Furthermore, the issue prevented all the liquidity providers from withdrawing all their liquidity.

Recommendation

Reorder the if-statements in `PoolHedger.updatePosition(...)` to ensure that `PoolHedger.sendAllQuoteToLP()` is always called before returning. This also ensures that an event is emitted when the net delta is updated.

Update

This issue was remediated in 80c739c.

5.5.3 `LyraGlobals.getSpotPrice(...)` should use Synthetix `isInvalid` flag (low risk)

LyraGlobals.sol#L349

Description

The `LyraGlobals.getSpotPrice(...)` function fetched the spot price of asset using the Synthetix `exchangeRates` function `rateAndUpdatedTime(...)` and validated it to ensure that the price is not stale according to the `stalePricePeriod`. The Synthetix `exchangeRates` contract exposes a similar function, `rateAndInvalid(...)`, which returns the rate along with an `isInvalid` flag. The invalid flag indicates whether the rate is stale according to Synthetix stale rate period, or if it has been flagged by the Chainlink aggregators. Since this flag is not fetched, it may be possible that an invalid rate is being used in the LyraGlobals.

Recommendation

The `exchangeRates.rateAndInvalid()` should be used in `LyraGlobals.getSpotPrice()`, with the returned `isInvalid` flag validated to be false.

Update

This issue was remediated in 39c1f7e.

5.5.4 Unsafe arithmetic used (informational)

OptionMarket.sol#L611, *LiquidityPool.sol#L289*

Description

In `OptionMarket._liquidateExpiredBoard()`, when working out the `amountToLiquidate` for a call to `liquidityPool.liquidateCollateral()`, unsafe arithmetic was used, as follows: `totalBoardLongCallCollateral + totalAMMShortCallProfitBase`. Similarly, the `value` to `withdraw()` in the `LiquidityPool`, was calculated as `(certificateData.liquidity * exitValue) / enterValue`. No explicit overflow or underflow scenarios were identified, which significantly lowers the risk of this issue.

Recommendation

As Solidity versions under 0.8.0 don't have over and underflow arithmetic protection by default, all arithmetic operations for unsigned integers should be replaced with the corresponding version in the SafeMath library.

Update

This issue was remediated in 39c1f7e.

5.5.5 Design comments (informational)

Actions to improve the functionality and readability of the codebase are outlined below.

Fix spelling and grammatical errors

Language mistakes were identified in the comments and revert messages in the codebase. Fixing these mistakes can help improve the end-user experience by providing clear information on errors encountered, and improve the maintainability and auditability of the codebase.

1. LiquidityCertificate.sol#L55: `addres` → `address`.
2. OptionMarket.sol#L627: `bord` → `board`.

Update

Both spelling errors were fixed in 39c1f7e.

Unused constant

The OptionMarket.lastRoundFinalised state variable was defined, but not used. It is recommended that it is removed.

Update

This unused constant was removed in 39c1f7e.

Gas optimizations

1. The requirement to check if the `boardId` is valid in the OptionMarket.openPosition should be moved to the beginning of the function call so that it can revert earlier.

The following functions should have their visibility restricted for additional gas savings:

2. LyraGlobals.setGlobals()
3. LyraGlobals.setGlobalsForContract()

Update

All gas optimizations were implemented in 39c1f7e.

Missing validation

The following function did not validate if the passed parameters were not the zero address:

1. The LiquidityCertificate constructor did not validate that the `_liquidityPool` address was not the zero address.

Update

The missing validation was implemented in 39c1f7e.