Lyra Avalon Smart Contract Audit

Lyra, 17 May 2022

1. Introduction

iosiro was commissioned by Lyra to conduct a smart contract audit of their Avalon release. The audit was performed by 2 auditors between 4 April and 28 April 2022, using 34 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 understand the smart contracts' risk exposure better and as a guide to improving the security posture of the smart contracts by remediating issues identified. The results of this audit reflect the in-scope source code reviewed at the time of the audit.

The purpose of this audit was to achieve the following:

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

Assessing the off-chain functionality associated with the contracts, for example, backend web application code, was outside of the 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 high risk from 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 Lyra smart contracts for their Avalon release. Lyra is a layer 2 options AMM allowing users to trade various options based on synthetic assets. Several features were introduced compared to the previous version of the system, many of which are detailed in the Design Specification below.

Audit findings

iosiro noted two high risk issues and several informational issues. One high risk issue related to an unsafe casting issue resulting in an attacker being able to drain the system of its base and quote assets, while the second high risk item related to an internal accounting issue. Overall, the system was found to operate as intended and the implementation was of a high standard.

Recommendations

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

- Remediating the issues identified in this report and performing a review to ensure that the issues were correctly addressed.
- Performing additional audits at regular intervals, as security best practices, tools, and knowledge change over time. Additional audits throughout 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 for the purposes of this audit.

3.1.1 Smart contracts

- Project name: Lyra
- Initial commit: bc0f77d72e0e26cc94c8ac3c0bdea955808c7240
- Final commit: bee3f72b9d5901d2e2174beb604d46b59c3a4dfb
- Files: LiquidityPool.sol, LiquidityTokens.sol, OptionGreekCache.sol, OptionMarket.sol, OptionMarketPricer.sol, OptionToken.sol, PoolHedger.sol, ShortCollateral.sol, SynthetixAdapter.sol

3.2 Methodology

The audit was conducted using a variety of techniques described below.

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 discover security issues that could be exploited. The coverage report of the provided tests as on the final day of the audit is given below.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	99.79	100	100	
LiquidityPool.sol	100	98.78	100	100	
LiquidityTokens.sol	100	100	100	100	
OptionGreekCache.sol	100	100	100	100	
OptionMarket.sol	100	100	100	100	
OptionMarketPricer.sol	100	100	100	100	
OptionToken.sol	100	100	100	100	
PoolHedger.sol	100	100	100	100	
ShortCollateral.sol	100	100	100	100	
SynthetixAdapter.sol	100	100	100	100	
contracts/lib/	100	100	100	100	
BlackScholes.sol	100	100	100	100	
GWAV.sol	100	100	100	100	
SimpleInitializeable.sol	100	100	100	100	

Overall, the code coverage of the unit and integration tests were expansive.

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. Static analysis results were reviewed manually and any false positives were removed. Any true positive results are 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 satisfactorily addressed, removing the risk it posed.

4. Design specification

The following section outlines the intended functionality of the system at a high level. This specification is based on the implementation in the codebase. Any perceived points of conflict should be highlighted with the auditing team to determine the source of the discrepancy.

Optimism

Lyra was designed to be deployed on Optimism OVM and integrated with Synthetix L2 to exchange assets and open short positions.

Compared to the previous release, shorting on L2 (SIP-135) was live at the time of the assessment. Given that Lyra relies on L2-specific Synthetix features such as no trade settlement, Lyra was still regarded as incompatible with Ethereum mainnet.

Option Markets and Option Types

The Option Market is a collection of listings, called *boards*, which a user can trade against. Boards are created by the Lyra team, with a board specifying an expiry time and a list of strike prices. The entire system is redeployed for each intended base and quote asset pair to trade against. Option positions are represented by ERC-721 tokens, which could either be split into multiple, smaller, positions or merged into a bigger position.

Compared to the previous Lyra version, Option Markets now support five types of options:

- Long Call
- Short Call (collateralized in the base asset)
- Short Call (collateralized in the quote asset)
- 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. However, compared to the previous version of Lyra, short options no longer need to be fully collateralized while undertaking the risk of liquidation.

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 some cutoff period, 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.

Partial Collateralization (https://leaps.lyra.finance/leaps/leap-18/)

Users taking out short options no longer need to meet the full capital requirement - instead, users are supplemented with the remaining capital from the liquidity pool to collateralize the option. Partially collateralized options take on the risk of liquidation, where options are subject to being liquidated if the position falls below some minimum collateral amount. Liquidations are permissionless, and incentivised by a portion of the liquidation penalty that is allocated when a position is liquidated. The liquidation penalty is further split between liquidity providers and security module stakers.

Universal Closing (https://leaps.lyra.finance/leaps/leap-18/)

The introduction of universal closing allows users to close positions that are very in-the-money or very out-of-the-money to collect profits or free up collateral respectively. These positions are penalized in a way that ensures the AMM has a positive expected value from these trades. These trades do not affect the baseline volatility, rather only the skew of the affected listing.

Liquidity pool

Liquidity for Option Market instances are provided by users depositing the quote asset, usually `sUSD`. A number of fungible `LiquidityTokens` are issued on deposit based on the NAV of the system. The `LiquidityTokens` are standard ERC-20 tokens and are transferable. Compared to the previous version of Lyra, users can withdraw their liquidity at any time subject to a short delay, as detailed below.

Anytime entry exit (https://leaps.lyra.finance/leaps/leap-16/)

In the previous Lyra version, option trading took place over rounds. In the Avalon release, liquidity providers will be able to deposit or withdraw funds at any time, subject to a short delay. Lyra plans to constantly list several expiries at a time, so to protect existing liquidity providers, users entering and exiting the system will need to wait for a cooldown period before their funds are added or withdrawn from the system.

Hedging

Hedging is a technique used by Lyra that allows the AMM to be neutral to the price movements of the underlying assets it holds. A permissionless `hedgeDelta` function is made available to hedge the exposure of the AMM. This function determines whether the AMM is net long or net short, and reduces or increases the relevant exposure needed to keep the system delta neutral. An interaction delay is set to prevent keepers from continually calling the `hedgeDelta` function to prevent the system from taking on too much exchange fees.

5. Detailed findings

The following section details the findings of the audit.

5.1 High risk

No identified high-risk issues were open at the conclusion of the review.

5.2 Medium risk

No identified medium-risk issues were open at the conclusion of the review.

5.3 Low risk

No identified low-risk issues were open at the conclusion of the review.

5.4 Informational

5.4.1 Design comments

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

Flash-loan NFT split

Users and other protocols integrating with Lyra should be aware that depositing `OptionTokens` in any vault that supports flash loans poses significant risks and could result in a loss of funds.

An attacker that is able to flash loan `OptionTokens`, could use the `split` functionality to create a new token containing most of the position size and collateral. Leaving the value of the original position significantly reduced.

Unprotected init

The `SynthetixAdapter` and `OptionMarketViewer` contracts' `init()` functions remain unprotected. These functions should be access-controlled to be in line with the other initializer functions throughout the codebase.

Failsafe transfers

The `LiquidityPool._transferQuote()` function, and other token transfer functions, should be capped to the contract balance to prevent the quote contract from reverting in the case of an insufficient balance. While we haven't identified such cases, we'd recommend adding it as a fail-safe.

5.4.2 Use precomputed contract addresses

Description

When deploying contracts, the contract address is calculated as the hash of the message signer's address and their nonce. Consequently, it is possible to precompute each contract's address during deployment. This allows the initialization pattern of the contracts to be reworked, so that the precomputed addresses are passed as constructor arguments, instead of arguments to separate initializer functions. The immutable keyword could then be used for properties that are intended to be constant.

Immutable variables reduce the number of storage slots used by a contract, which in turn reduces gas fees of any subsequent transactions.

Further information on the `immutable` keyword can be found here:

https://blog.soliditylang.org/2020/05/13/immutable-keyword/

Recommendation

It should be investigated whether modification of the deployment scripts to support precomputation of each contract's address is achievable and maintainable. Subsequently, a large number of the state variables set in the various contract initializers can be reworked to make use of the `immutable` keyword.

5.4.3 Dangerous assumption regarding position states

OptionToken.sol#L458, *OptionToken.sol#L409*, *OptionToken.sol#L286*

Description

`OptionToken.settlePositions()`, `OptionToken.merge()` and `OptionToken.split()` did not explicitly validate the state of each position, but rather relied on `ownerOf` and `_isApprovedOrOwner` functions to revert for burned tokens.

In particular, `OptionToken.settlePositions()` relied on the `getPositionsWithOwner()` function call within `ShortCollateral.settleOptions()` to prevent settlement of invalid positions.

Recommendation

As a defense-in-depth measure, the state of each position should be explicitly validated in the aforementioned functions. This would be consistent with the validation performed in `adjustPosition()` and `addCollateral()`.

As an example, the following validation should be performed:

```
if(position.state != PositionState.ACTIVE) revert InvalidPositionState();
```

5.5 Closed

5.5.1 Unsafe cast resulting in loss of funds (High risk)

OptionMarket.sol#L443

Description

In the `OptionMarket` contract, a user can increase the collateral of their short position by calling `addCollateral()`, which accepts the position's ID, as well as the amount of collateral to add as a `uint` (`amountCollateral`). This method first calls `OptionToken.addCollateral()`, which updates the position's collateral amount by adding `amountCollateral` to the existing value. After this, a call is made to `_routeUserCollateral()`, which will either request a deposit from the position holder to supplement their collateral, or withdraw funds to the position holder in the event of excess funds. This operation is determined by the sign of the collateral amount passed to `_routeUserCollateral()`.

However, when calling `_routeUserCollateral()`, `amountCollateral` is cast to an `int` which can result in a condition whereby the target position's collateral is updated, but a negative value is passed to `_routeUserCollateral()`, indicating a withdrawal of funds to the position holder. Exploiting this issue allows an attacker to steal funds in the custody of the `ShortCollateral` contract. Additionally, during this process the position's collateral would be updated to some large value that could allow the attacker to further drain the remaining funds by updating the position's collateral through the `openPosition()` method in the `OptionMarket` contract.

An example attack is as follows:

1. The attacker opens a short position of type `SHORT_CALL_BASE` or `SHORT_CALL_QUOTE`, depending on whether they want to deplete the base or quote asset. The attacker, in this example, specifies 0.1 `ether` as the `setCollateralTo` value of the trade parameters. After this operation, the position's collateral is 0.1 `ether`.
2. The attacker calls `addCollateral()` in `OptionMarket`, specifying the newly created position's ID, and the amount `0xFFF...FFF0000000000000000` as `amountCollateral`. The `OptionToken` contract will add this to the position's

collateral, resulting in the position's collateral being `0xFFF...FFF016345785d8a0000`, avoiding an arithmetic overflow and setting the collateral to a very large value.

3. `_routeUserCollateral()` is called, whereby the value of `amountCollateral` is casted to an `int`, which is the value -18446744073709551616 (approximately -18.4 `ether`). Due to the negative sign, `_routeUserCollateral()` will transfer an amount of 18.4 `ether` worth of the asset to the attacker, already netting the attacker a significant amount of funds.
4. As the position's collateral is a large number, which is conceivably larger than the total amount of the target asset the `ShortCollateral` contract holds, the attacker can drain the remaining funds by decreasing their position's collateral by the amount of the asset still in custody of the `ShortCollateral` contract. The `OptionMarket` will recognize that the position holder has reduced their collateral and transfer the difference to the position holder, sending all remaining funds to the attacker.

A proof of concept was developed to illustrate how the vulnerabilities could be exploited and is available at the following location:

https://gist.github.com/AshiqAmien/3929f6748b051707246494492001f789

Recommendation

As for remedial action, the `addCollateral()` method in the `OptionMarket` contract should make use of OpenZeppelin's `SafeCast` library to ensure the transaction is reverted when `amountCollateral` might result in a negative value after casting. The `SafeCast` library includes the `toInt256()` function that accepts an `uint` to be casted to an `int`. This recommendation should be applied to other instances in the codebase where unsafe casting is used, such as the unsafe cast of `setCollateralTo` in `OptionToken.adjustPosition()`.

Update

This issue was remediated by using OpenZeppelin's `SafeCast` library to cast `uint` values to `int`.

5.5.2 Donated tokens cause incorrect NAV calculation resulting in loss of funds (High risk)

PoolHedger.sol#L166, *PoolHedger.sol#L185*

Description

The `PoolHedger` contract used the balance of the `baseAsset` to determine whether to include the short balance of the system. As it was possible to directly transfer `baseAsset` to the `PoolHedger` contract, the net asset value calculation of the system could be manipulated. An attacker could exploit this to reduce the cost to mint liquidity tokens or inflate it when withdrawing, depending on the current hedge of the system.

A proof of concept was developed to illustrate how the vulnerabilities could be exploited and is available at the following location:

https://gist.github.com/AshiqAmien/e6c3c9976c59e09a27a046650f6c9bee

Recommendation

The long balance should be calculated as the sum of both the `baseAsset` balance of the contract as well as the current short balance.

```
function getHedgingLiquidity(ICollateralShort short, uint spotPrice)
    external
    view
    returns (uint pendingDeltaLiquidity, uint usedDeltaLiquidity)
  {
    // Get capped expected hedge
    int expectedHedge = getCappedExpectedHedge();

    // Get current hedge
    (uint shortBalance, uint shortCollateral) = getShortPosition(short);
    uint longBalance = baseAsset.balanceOf(address(this));

-    int currentHedge = longBalance > 0 ? int(longBalance) :
-int(shortBalance);

+   int currentHedge = int(longBalance) - int(shortBalance);

function getCurrentHedgedNetDelta() external view returns (int) {

    SynthetixAdapter.ExchangeParams memory exchangeParams =
synthetixAdapter.getExchangeParams(address(optionMarket));


    uint longBalance = baseAsset.balanceOf(address(this));

-    if (longBalance > 0) {

-      return int(longBalance);

-    }

-    (uint shortBalance, ) = getShortPosition(exchangeParams.short);

-    return -int(shortBalance);

+    return int(longBalance) - int(shortBalance);

  }
```

Update

The issue was resolved in commit bda73fb.

The implementation of `PoolHedger.getCurrentHedgedNetDelta()` was not modified and could still report an incorrect amount, however, it could not be exploited as the function was not called anywhere within the system. There is still some risk to systems integrating with Lyra that might rely on the function.

5.5.3 Liquidity providers exposed to the risk of dynamic exchange fees (low risk)

OptionMarketPricer.sol#L295

Description

Synthetix introduced Dynamic Exchange Fees (SIP-184) for exchanges shortly before the audit. In periods of high volatility, an additional exchange fee is levied in addition to the base exchange fee. Under very specific circumstances, it could be possible that the `spotPriceFee` and other fees levied on the options are insufficient to cover the inflated exchange fees. The liquidity providers would then ultimately provide the necessary liquidity to perform the exchange.

Recommendation

The `spotPriceFee` could be adapted to scale depending on the current exchange fee rates, as opposed to being a static rate. However, since the `PoolHedger` can also result in assets being exchanged, a limit on the exchange fee should be investigated to protect liquidity providers.

Update

A mechanism to bypass exchanging of assets during periods of inflated exchange fees was introduced in commit 68db076.

In the event that exchange fees are above the configured threshold, the desired `baseAsset` and `quoteAsset` balances will be tracked without exchanging. On subsequent calls, the net difference will be exchanged.

5.5.4 Design comments (informational)

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

Missing reentrancy guards

There are unprotected external functions that don't use a reentrancy guard, such as `LiquidityPool.updateCBs()`, `OptionToken.split()` and `OptionToken.merge()` functions. While no reentrancy risk was identified, it was brought up to be consistent with the rest of the unprotected external functions that have a reentrancy guard.

Code refactor

The following functions should be renamed to be in line with Solidity naming conventions:

- `reclaimInsolvency -> _reclaimInsolvency` (ShortCollateral.sol)
- `sendLongCallProceeds -> _sendLongCallProceeds` (ShortCollateral.sol)
- `sendShortCallBaseProceeds -> _sendShortCallBaseProceeds` (ShortCollateral.sol)
- `sendShortCallQuoteProceeds -> _sendShortCallQuoteProceeds` (ShortCollateral.sol)
- `sendShortPutQuoteProceeds -> _sendShortPutQuoteProceeds` (ShortCollateral.sol)
- `getPoolHedgerLiquidity -> _getPoolHedgerLiquidity` (LiquidityPool.sol)
- `getGWAVVolWithOverride -> _getGWAVVolWithOverride` (OptionGreekCache.sol)
- `isPriceMoveAcceptable -> _isPriceMoveAcceptable` (OptionGreekCache.sol)
- `isUpdatedAtTimeStale -> _isUpdatedAtTimeStale` (OptionGreekCache.sol)
- `timeToMaturitySeconds -> _timeToMaturitySeconds` (OptionGreekCache.sol)
- `getSecondsTo -> _getSecondsTo` (OptionGreekCache.sol)
- `sendAllQuoteToLP -> _sendAllQuoteToLP` (PoolHedger.sol)

Gas Optimizations

- LiquidityPool.sol#L293 – `burnAmount` can be moved down after the short circuit for `totalTokensBurnable == 0`.

- OptionMarket.sol#L824 – `OptionMarket._settleExpiredBoard()` should use a cached value for `strikeToBaseReturnedRatio[strike.id]` and only set and read the slot once.
- LiquidityPool.sol#L209 and LiquidityPool.sol#L233 – The `require(beneficiary != address(0), "invalid beneficiary");` check can be done first in `initiateDeposit` and `initiateWithdraw`.