Lyra

PPTSA Contracts

Security Assessment Report

Version: 2.0

August, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Lyra smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Lyra smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Lyra smart contracts.

Overview

At a high level, Lyra maintains a vault which trades on behalf of depositors. The vault owns a sub-account in the Lyra matching system and registers a session key so that it can trade on the exchange. Shares represent a portion of that sub-account (plus any excess funds just deposited or being withdrawn).

The `PPTSA` contract applies four trading strategies into a principal protected vault, allowing for parameterised protection on principal during short and long call spreads, as well as short and long put spreads.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Lyra's v2-matching repository.

The scope of this time-boxed review was strictly limited to changes in PR#53 at commit 1c78b99.

This included changes to the following files:

- `src/tokenizedSubaccounts/`
  - `CCTSA.sol`
  - `CollateralManagementTSA.sol`
  - `PPTSA.sol`

Note: third party libraries and dependencies, such as OpenZeppelin, were excluded from the scope of this assessment.

Approach

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 4 issues during this assessment. Categorised by their severity:

- Low: 3 issues.
- Informational: 1 issue.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Lyra smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
PPTSA-01	Critical Assumptions May Break Intended PPTSA Protections	Low	Resolved
PPTSA-02	RFQ Fee Calculation Errors	Low	Resolved
PPTSA-03	Open Spread Loss Calculation Error	Low	Resolved
PPTSA-04	Miscellaneous General Comments	Informational	Closed

PPTSA-01	Critical Assumptions May Break Intended PPTSA Protections		
Asset	`PPTSA.sol`		
Status	**Resolved:** See PPTSA-01		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Several critical assumptions are made by the `PPTSA` contract about the `v2-core` dependencies, which may be broken through various means.

This may cause serious unintended outcomes such as allowing trades that should not be allowed, or preventing withdrawals.

The following assumptions may be bypassed or broken:

1. **Only `PPTSA` contract is able to influence option balances**

   The following snippet on line [**528**] is responsible for verifying options balances, which is then used to determine if more open spreads can be purchased, or prevent full withdrawal of collateral:

   `balances[i].asset == _getPPTSAStorage().optionAsset && balances[i].balance > 0`

   This assumes that only the `PPTSA` contract is able to directly influence the balance of the `PPTSA` sub-account option balance. However, this is not the case - the option balance can also be updated directly by the `TSA` sub-account manager, as seen defined in the following modifier:

   ```
   function _isApprovedOrOwner(
     address spender,
     uint accountId
   ) internal view override returns (bool) {
     if (super._isApprovedOrOwner(spender, accountId)) return true;

     // check if caller is manager
     return address(manager[accountId]) == msg.sender;
   }
   ```

   A manager may be able to send a single option spread in either the positive or negative balance, which would throw off the accounting assumptions.

   Withdrawals would not be possible if a single positive open spread is owned by the TSA sub account.

2. **Cash is liquid**

   During withdrawals, the `PPTSA` contract validates that enough base collateral is available and `cashBalance >= 0`.

   If cash is positive, withdrawals are possible since cash balance is assumed to be liquid. If cash is negative, the entire cash balance of an account is checked to see if there is enough base balance after withdrawal to cover an allowed amount of negative cash debt.

   However, the liquidity of cash is subject to the `SecurityModule` ability to pay liquidators, as seen in the following code block:

```
function getIsWithdrawBlocked() external view returns (bool) {
  if (totalInsolventMM > 0 && smAccount != 0) {
    int cashBalance = subAccounts.getBalance(smAccount, cash, 0);
    // Note, negative cash balances in sm account will cause reverts
    return totalInsolventMM > cashBalance.toUint256();
  }
  return false;
}
```

This function in the `DutchAuction` contract prevents withdrawals in the `CashAsset` contract if the sub account of the `SecurityModule` does not have enough cash to cover `totalInsolventMM`.

If this is the case, no cash withdrawals are possible until either the `totalInsolventMM` has decreased, or the amount of funds held by the SM has increased.

During this time however, the `PPTSA` allows for full withdrawal of the base wrapped asset, as it assumes the cash balance at the time is liquid. Therefore, it is possible, that the total value locked in the `PPTSA` contract decreases substantially, whilst assuming cash is of a certain value. Once withdrawals are possible and socialising losses has stabilised, the vault itself would therefore be in a state where cash valuation is lower than anticipated.

3. **Base balance is positive**

   The function `_verifyWithdrawAction()` makes an assumption that `baseBalance` is always positive (i.e. the `abs()` value of the `int` is used when obtaining the balance).

   However, in the case that the base asset is cash, it is possible that the `cashBalance` is negative, whilst the `_getSubAccountStats()` function returns a positive value for the `baseBalance`.

   This may lead to situations where the value may be extracted out of the `TSA`.

Recommendations

Implement stricter rule sets within the `PPTSA` contract to ensure that the key assumptions made across multiple contracts are correct and do not contradict each other.

Examples of suggested rule sets or stronger invariants might be:

1. A separate function to check for open positive and negative balances of open spreads, performing reconciliation to ensure that there is never a non zero sum
2. If cash can be frozen and loses socialised, implement additional checks for these states within the `PPTSA` contract
3. If the base balance can only be positive, reject assets that may have negative balances

Resolution

This set of issues was resolved in the commit 059408f.

PPTSA-02	RFQ Fee Calculation Errors		
Asset	`PPTSA.sol`		
Status	**Resolved:** See PPTSA-02		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The function `verifyRFQFee()` uses `totalLegAmount` to calculate the `maxAllowedTradeFee` by multiplying it with the base price instead of the actual option trade price.

Depending on the base price in comparison to the options asset trade price, this could result in healthy trades reverting or invalid trades passing.

On line [462], consider the `maxAllowedTradeFee` calculation below:

```
uint maxAllowedTradeFee = totalLegAmount.multiplyDecimal(\$.ppParams.rfqFeeFactor).multiplyDecimal(\_getBasePrice());
```

It conducts the following operations:

1. The higher strike is added to the lower, resulting in `totalLegAmount`
2. The result is multiplied by the `rfqFeeFactor`
3. The result is multiplied by the `_getBasePrice()`

The higher strike added to the lower strike is not the actual option trade price and, as a result, the resulting fee calculation is incorrect.

Recommendation

The testing team recommends switching the base price for the total trade cost, and then multiplying this by the `rfqFeeFactor`.

Resolution

This issue was resolved in the commit 059408f.

PPTSA-03	Open Spread Loss Calculation Error		
Asset	`PPTSA.sol`		
Status	**Resolved:** See PPTSA-03		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `maxLossOfOpenOptions` variable prevents new RFQ option strategies from opening if previously opened strategies exceed potential max losses.

However, this check is calculated on the assumption that the `strikeDiff` cannot change. If `strikeDiff` decreases substantially in future actions verification, max losses will be understated, allowing trades when they in fact should be rejected.

On line [**437**], the following calculation is based on current `strikeDiff` of the RFQ action being verified:

```
uint maxLossOfOpenOptions = openSpreads.multiplyDecimal(strikeDiff);
```

Previously opened spreads may have had a different `strikeDiff` as this value can be updated with calls to `setPPTSAParams()`. If there are huge decreases, this will undermine checks to protect `PPTSA` against undercollateralised strategies.

Additionally, it is worth mentioning that `openSpreads` is calculated only on positive balance amounts. Depending on the type of open spread, it is possible that the higher strike price is a negative amount, which will mean the total cost is potentially understated (depending on long or short positions on that call or put spread). As such, there may be the potential for further inaccuracies on this value.

Recommendation

Consider implementing accounting at the sub-account level that keeps track of current open spread trade costs and leveraging these values directly, as opposed to making assumptions and rough calculations within the `TSA` contract during `signActionData()`.

Resolution

This issue was resolved in the commit 059408f.

PPTSA-04	Miscellaneous General Comments
Asset	PPTSA.sol
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **`setPPTSAParams()` does not check if `minMarkValueToStrikeDiffRatio > 0`**
   Parameter verification reverts if
   `pptsaParams.minMarkValueToStrikeDiffRatio > pptsaParams.maxMarkValueToStrikeDiffRatio`
   However, this does not exclude the possibility that `pptsaParams.minMarkValueToStrikeDiffRatio == 0`.
   This subsequently allows a `markValueToStrikeDiffRatio == 0` to pass during internal function call `_validateTradeDetails()`.
   Consider implementing restrictive lower bounds for `minMarkValueToStrikeDiffRatio`.
2. **Overloaded Variable Name Usage** Several cases exist with confusing variable names:
   (a) line [**256**]:
   `uint remainingBaseBalance = (baseBalance - withdrawalAs18Decimals).multiplyDecimal(_getBasePrice())`
   has `baseBalance` multiplied by `_getBasePrice()`, which then returns a `remainingBaseBalance`.
   It is important to note what units are being used in calculations, if base balance refers to amount of tokens, then `remainingBaseBalance` may be better named as `remainingBaseValue`.
   (b) line [**439**]:
   `uint baseBalanceAtSpotPrice = baseBalance.multiplyDecimal(_getBasePrice());` has a `baseBalance` multiplied by `_getBasePrice()` and returns a `baseBalanceAtSpotPrice`.
   Have clearer and consistent differentiation between `uint` types: if referring to total value of the `baseAmount`, a `baseValue` name may be better suited and avoid confusion.
   (c) line [**533**]: `openSpreads` refers to open spreads, however, it more accurately refers to `openPositiveSpreads`.

   Where applicable, consider variable name changes to accurately reflect the units and actions taken within existing sequence of instructions.
3. **`PPT_InvalidParams` is an ambiguous error**
   `PPT_InvalidParams` on line [**336**] and line [**366**] is ambiguous, clearer options might include `PPT_InvalidTradeLength` and `PPT_InvalidMakerAsset`.
   Parameters within the TSA contract tend to refer to those set within the `setPPTSAParams()` function.
   Consider renaming the custom reverts to be more descriptive.
4. **`setPPTSAParams()` deployment scripts do not properly use `setCollateralManagementParams()`**
   The files `deploy-tsa.s.sol` and `upgrade-tsa.s.sol` do not accurately reflect the correct usage of `setPPTSAParams()` and `setCollateralManagementParams()` functions.
   The testing team recommends maintaining the deployment scripts with the current state of mainnet contracts to avoid compatibility issues and introduction of upgrade vulnerabilities.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team addressed above recommendations, where applicable, in the commit 059408f.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `Forge` framework was used to perform these tests and the output is given below.

```
Ran 2 tests for test/BaseTSATest.t.sol:BaseTSATest
[PASS] test_full_deposit_workflow() (gas: 1569771)
[PASS] test_initiateDeposit_BaseTSA() (gas: 1117127)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 20.83ms (2.63ms CPU time)

Ran 19 tests for test/PPTSA.t.sol:PPTSATest
[PASS] testDepositThenWithdraw() (gas: 9275881)
[PASS] testGetPPTSAValues() (gas: 6862315)
[PASS] testSetInvalidPPTSAParams() (gas: 6562646)
[PASS] testTradeActionLongCallPPTSA() (gas: 10865844)
[PASS] testTradeActionLongPutPPTSA() (gas: 10865821)
[PASS] testTradeActionShortCallPPTSA() (gas: 10865758)
[PASS] testTradeActionShortPutPPTSA() (gas: 10845924)
[PASS] testTradeRFQOptionLongCallVaultMaker() (gas: 11193196)
[PASS] testTradeRFQOptionLongCallWithLargeNegativeCash() (gas: 9416547)
[PASS] testTradeRFQOptionLongPutVaultMaker() (gas: 11197917)
[PASS] testTradeRFQOptionRevertInvalidHighStrikeAmount() (gas: 9267809)
[PASS] testTradeRFQOptionRevertInvalidOptionDetails() (gas: 9234947)
[PASS] testTradeRFQOptionRevertInvalidParamsMoreThanTwoTrades() (gas: 9280285)
[PASS] testTradeRFQOptionRevertStrikePriceOutsideOfDiff() (gas: 9268094)
[PASS] testTradeRFQOptionRevertTradeDataDoesNotMatchOrderHash() (gas: 9218887)
[PASS] testTradeRFQOptionRevertTradeTooLarge() (gas: 9285105)
[PASS] testTradeRFQOptionShortCallVaultMaker() (gas: 11207697)
[PASS] testTradeRFQOptionShortCallVaultTaker() (gas: 11206835)
[PASS] testTradeRFQOptionZeroAmounts() (gas: 9265775)
Suite result: ok. 19 passed; 0 failed; 0 skipped; finished in 33.82ms (125.11ms CPU time)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].