LYRA

V2-Matching Contracts

Security Assessment Report

Version: 2.0

July, 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 Shares represent a portion of that sub-account (+ any excess funds just deposited/being withdrawn).

The first use case of the new contracts will be a yield bearing delta neutral trade, where users will deposit yield bearing staked ETH (i.e. wstETH). An equivalent amount of short perps will be opened against the collateral to make it delta neutral, and to earn the additional funding on the perps.

Security Assessment Summary

Scope

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

The scope of this time-boxed review was strictly limited to the following files at diff 772b45e...e5be950:

- `src/tokenizedSubaccounts/BaseOnChainSigningTSA.sol`
- `src/tokenizedSubaccounts/BaseTSA.sol`
- `src/tokenizedSubaccounts/LRTCCTSA.sol`
- `src/tokenizedSubaccounts/TSA.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 a 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 7 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 3 issues.
- Informational: 3 issues.

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
LYR2-01	Processing Non-Existent Withdrawal Requests	High	Resolved
LYR2-02	Checks-Effects-Interactions Pattern Violations In `BaseTSA` Contract	Medium	Resolved
LYR2-03	Double-Spent Deposits Can Lead To Under-Collateralised TSA	Medium	Resolved
LYR2-04	Failed Token Transfers Will DoS Withdrawals	Medium	Resolved
LYR2-05	Delaying Withdrawal By Flooding The Withdrawal Queue	Informational	Resolved
LYR2-06	Compounding Fee Collection May Drain User Funds	Informational	Closed
LYR2-07	Miscellaneous General Comments	Informational	Closed

LYR2-01	Processing Non-Existent Withdrawal Requests		
Asset	`BaseTSA.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Anyone can call `processWithdrawalRequests()` and process non-existent withdrawal requests, blocking future legitimate withdrawal requests.

The function `processWithdrawalRequests()` doesn't check if `$.queuedWithdrawalHead` value is less than the last created withdrawal request. As per line [**279**], `$.queuedWithdrawalHead` is used as a `withdrawalId` of the withdrawal request `queuedWithdrawals`. As such, if `$.queuedWithdrawalHead` is greater than the last created request, the function will not revert, but instead all fields of `queuedWithdrawals` will be set to zero.

Therefore, if the `depositToken` doesn't revert on transferring to `address(0)`, it is possible to process a non-existent request. As a result, newly created requests may not be processed due to the mismatch between `$.queuedWithdrawalHead` and the `withdrawalId` of the last created request.

Recommendations

Ensure that `$.queuedWithdrawalHead <= nextQueuedWithdrawalId + limit` or check that `amountShares` of a request is greater than 0.

Resolution

The finding has been resolved at commit 5f01362.

LYR2-02	Checks-Effects-Interactions Pattern Violations In `BaseTSA` Contract		
Asset	`BaseTSA.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

`BaseTSA` contracts implements some critical functions that violate the *Checks-Effects-Interactions* (CEI) pattern. Under certain external calls this may create unexpected changes to Lyra accounting.

Most notably in `processWithdrawalRequests()` on line [**304**], the token transfer happens before the state update, which allows an attacker to re-enter the function and drain the contract if the token implements an `_afterTokenTransfer()` (or equivalent) hook.

```
$.depositAsset.transfer(request.beneficiary, requiredAmount);

uint sharesRedeemed = request.amountShares;

$.totalPendingWithdrawals -= sharesRedeemed;
request.amountShares = 0;
request.assetsReceived += requiredAmount;
```

CEI violations also occur in:

- `processWithdrawalRequests()` on line [**291**].
- `initiateDeposit()` on line [**180**].

Recommendations

Restructure the implementation of these functions to follow the *Checks-Effects-Interactions* pattern, i.e. perform checks, then update the state (effects), and only after that interact with external contracts (interactions).

Otherwise, consider using a re-entrancy guard.

Resolution

The finding has been resolved at commit adad796.

LYR2-03	Double-Spent Deposits Can Lead To Under-Collateralised TSA		
Asset	`BaseTSA.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `BaseTSA` contract includes a two step deposit process. The first step includes `initiateDeposit()`, the second includes a shareKeeper or unprivileged user after some timelock calling `processDeposit()`. Due to discrepancies in accounting logic, it is possible to process the same deposit twice under specific circumstances. This may lead to an under-collateralised TSA contract where pending deposits are stolen from other users entering the vault.

The `BaseTSA` contract handles the processing of withdrawals as follows:

```
if (request.sharesReceived > 0) {
  revert BTSA_DepositAlreadyProcessed();
}
uint shares = _getSharesForDeposit(request.amountDepositAsset);

request.sharesReceived = shares;
$.totalPendingDeposits -= request.amountDepositAsset;
```

If `sharesReceived` returns 0, this same deposit can be processed again. Assuming 0 shares were minted for a diluted user, the actual transferred underlying asset is then socialised to all other participants in the protocol. These users are able to fully withdraw all their shares, meaning the protocol contains no TVL.

At this stage, if a user X deposits an underlying amount larger or equal to the originally diluted user amount, the diluted user is then able to reprocess their deposit obtaining non-zero shares. At this stage, user X is unable to finalise the processing of their deposit since `totalPendingDeposits < request.amountDepositAsset`.

Recommendations

Maintain a unique identifier mapping for deposits indicating a deposit has been redeemed, instead of relying on `sharesReceived`.

Resolution

The finding has been resolved at commit 5f01362.

LYR2-04	Failed Token Transfers Will DoS Withdrawals		
Asset	`BaseTSA.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

As the withdrawal requests are processed in the queue, it is possible that a single withdrawal in the queue gets stuck, preventing all other requests from processing.

The token transfer can fail for several reasons. If this happens, all requests that were created after a request that fails would not be processed.

Reasons for the failure or stuck state include:

- A token implements blacklisting, such as USDT, which will revert if the transaction is sent to blacklisted user.
- An amount of 0 is used as the transfer amount to the beneficiary.
- A token with transfer related hooks (e.g. `_beforeTokenTransfer()` or `_afterTokenTransferreverting()` or consuming all gas.

Recommendations

Change the function `processWithdrawalRequests()` to accept `withdrawalId` instead of the number of requests to process.

Alternatively, allow a privileged user to discard withdrawals that compromise withdrawal integrity.

Resolution

The finding has been resolved at commit 5f01362.

LYR2-05	Delaying Withdrawal By Flooding The Withdrawal Queue
Asset	`BaseTSA.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

As there is no minimum amount required in the function `requestWithdrawal()`, a malicious user could flood the withdrawal queue with large number of requests, each of a small withdrawal amount.

This could potentially delay the processing of subsequent withdrawal requests.

Note, as the economic feasibility of carrying out this attack is implausible, this finding is rated as informational only.

Recommendations

Implement a configurable minimum withdrawal amount in `requestWithdrawal()`.

Resolution

The finding has been resolved at commit 5f01362.

LYR2-06	Compounding Fee Collection May Drain User Funds
Asset	`BaseTSA.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The contract `BaseTSA` collects fees from users that depend on two factors: time since last collection, and number of shares currently in circulation. This means that fees can accrue, based on the fees already accrued by the fee recipient.

Fees accrue in the TSA contracts as follows:

```
uint timeSinceLastCollect = block.timestamp - $.lastFeeCollected;
uint percentToCollect = timeSinceLastCollect * $.tsaParams.managementFee / 365 days;
uint amountCollected = totalShares * percentToCollect / 1e18;

_mint($.tsaParams.feeRecipient, amountCollected);
```

However, `totalShares` increases with each fee collection, minted to the `feeRecipient` account. This means that fees themselves will accrue additional fees over time. Though compounding might be an expected behaviour, this does not prevent a fee recipient from using a separate account to inflate share numbers.

Furthermore, inflation and fee generation doesn't require active protocol usage to generate fees. If a fee recipient was to use a second account and own a large share holding of the `BaseTSA` contract, they effectively could increase the fees charged not only on their account, but also to the other share holders.

A maximum fee of 2 percent compounding on 5,000 shares, would take approximately 35 years to dilute the shares to 10,000, half their original value with no PnL. However, this assumes the fee recipient has made no initial deposits. If the fee recipient adds an additional 95,000 shares, which may be from a separate account, then the vault has total 100,000 shares. This will take approximately 3 years to dilute the value to 105,000 shares. If the fee recipient removes their amount, the vault is left with 10,000 shares. 5,000 belonging to the original depositors, and 5,000 belonging to the `feeRecipient` from fee calculations.

Compounding fees produce inevitable risks where users with significant capital at their disposal are able to consume other account values at increasing rates over time. Furthermore, these risks are amplified by charging fees without requiring protocol interaction.

Recommendations

Investigate alternative fee mechanisms that avoid minting compounded fees.

If compound fees are necessary, consider introducing explicit protections and agreements with fee recipients that exclude them from colluding with whales, along with documentation and warnings for customers.

Furthermore, it is highly advised to consider protocol interactions as fee capable events, this can prevent large holders consuming fees without actively partaking in the protocol. In the event interest should be charged, this can then be done on top of fees generated as a separate calculation over time.

Resolution

The finding has been deemed as a false-positive after discussion with the development team.

LYR2-07	Miscellaneous General Comments
Asset	All contracts
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. **`uint256/int256` instead of `uint/int`**
   Related Asset(s): tokenizedSubaccounts/*.sol
   For non-ambiguity and consistency, use `uint256/int256` instead of `uint/int`
   Consider changing `uint/int` to `uint256/int256` everywhere throughout the code.
2. **Unnecessary Argument In `DepositProcessed` Event**
   Related Asset(s): BaseTSA.sol
   The argument `success` in the event `DepositProcessed` is unnecessary as it always evaluates to `true`.
   Remove this argument from the event.
3. **Redundant Check**
   Related Asset(s): CCTSA.sol
   The function `_verifyFee()` check for the fees using `require` statement, but also performs the same check using `if revert` statement.
   Remove one of the redundant checks.
4. **Stealth Deposits Deflate Share Value**
   Related Asset(s): BaseTSA.sol
   Vault-like contracts can be subject to inflation attacks via stealth deposits of the underlying asset. This involves a second depositor being front-run by the first, with a direct transfer larger than the second depositor. However, due to some scaling, this requires an amount of `1e36` to effectively round the users depositing 1 ETH down to zero shares. Though the attack is impractical, it is still possible, and with varying underlying values attached to the shares it may make the attack plausible.
   Standard defences against this attack include the trusted first depositor, scaling (being used) and reverting zero mints. Additional defence worth considering is only calculating share price based on underlying funds submitted through the `initiateDeposit()` function. Though rounding is still possible in that event, but it requires many transactions, as opposed to a singular transaction.
5. **Vault-like Contracts Missing Explicit Rounding Direction Calculation**
   Related Asset(s): BaseTSA.sol
   With vault-like contracts, a lot of unexpected rounding errors can occur, sometimes if denominators are rounded down in one place, it may leave latter operations to be larger than expected (almost rounding up).
   The team is advised to use math libraries that allow for explicit rounding directions to be elected.

6. **Documentation for Lyra Matching**

   Related Asset(s): *.sol

   At the time of testing, there was no detailed documentation available describing the V2-matching system all of its intricacies.

   We recommend finalizing documentation for the benefit of future security assessors and users of the platform.

7. **Reduce complexity where possible**

   Related Asset(s): *.sol

   The V2-matching contract is highly coupled with various aspects of the V2-core contracts. Consider the following simple ways to reduce code complexity:

   - Avoid including multiple contracts in the same file where possible.
   - Avoid overloaded terminology. Functions like `_getSharesToWithdrawAmount` could be clearer as `_getSharesToAsset` as `asset` is a widely accepted unit of measurement in vault-like contracts corresponding to underlying token amounts. Likewise worth distinguishing between things like asset, wrappedDepositAsset and cash. Good terminology for the CCTSA contract, might include `underlyingTokens` or `settlementToken`.
   - Reduce the unnecessary use of type casting. For instance, the following statement `(convertedMtM + depositAssetBalance.toInt256()).abs()` commits two type cast conversions, where only one is needed.

Recommendations

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

Resolution

The development team acknowledged above comments and actioned the ones deemed relevant at commit 5f01362.

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 12 tests for test/BaseTSATest.t.sol:BaseTSATest
[PASS] test_approveModule() (gas: 77063)
[PASS] test_collectFee() (gas: 930919)
[PASS] test_full_deposit_workflow() (gas: 1584644)
[PASS] test_initiateDeposit_BaseTSA() (gas: 1094621)
[PASS] test_processDeposit() (gas: 982483)
[PASS] test_processDeposit_unprocessedDeposit_poc() (gas: 2332074)
[PASS] test_processDeposits() (gas: 1209645)
[PASS] test_processWithdrawalRequests12() (gas: 1181386)
[PASS] test_processWithdrawalRequests_partial() (gas: 2895095)
[PASS] test_processing_non_existant_withdrawal_request_poc() (gas: 8960456)
[PASS] test_reentrancy_poc() (gas: 10112394)
[PASS] test_requestWithdrawal() (gas: 1108384)
Suite result: ok. 12 passed; 0 failed; 0 skipped; finished in 35.75ms (110.77ms CPU time)

Ran 2 tests for test/CCTSA.t.sol:CCTSATest
[PASS] test_tradeAction12() (gas: 4081409)
[PASS] test_tradeAction_isBid() (gas: 3118372)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 38.56ms (48.65ms CPU time)

Ran 2 test suites in 144.76ms (74.30ms CPU time): 14 tests passed, 0 failed, 0 skipped (14 total tests)
```

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.

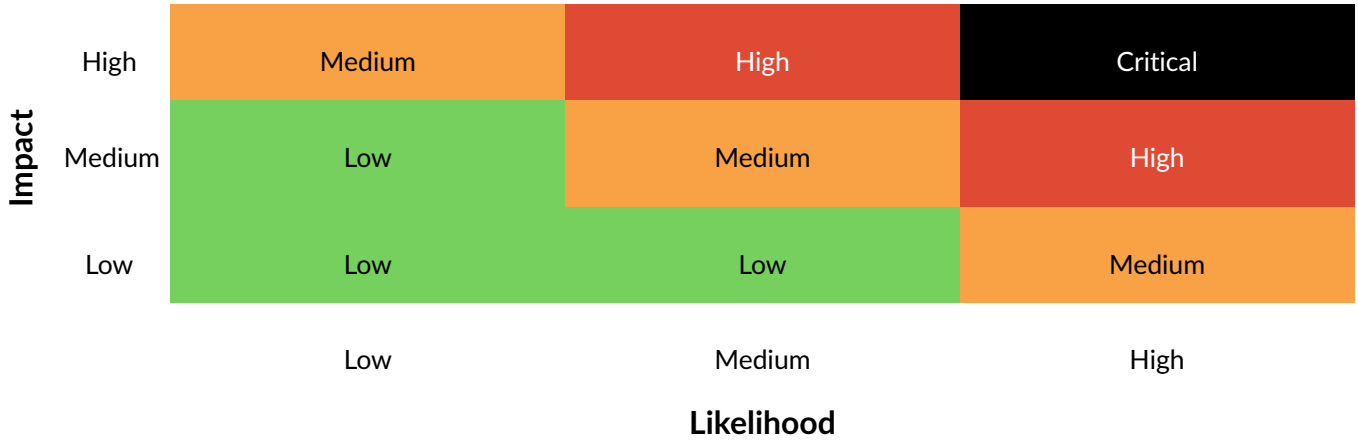

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].