Lyra Finance

Lyra V2

Smart Contract Security Review

Version: 2.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Lyra v2 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 v2 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 v2 smart contracts.

Overview

Lyra protocol is a decentralized financial technology protocol designed for options trading, allowing users to optimize trading strategies, hedge risks, and earn yield. The protocol features three main components: managers, liquidations, and cash.

1. **Managers –** There are two types of managers in the Lyra protocol: The Portfolio Margin Risk Manager (PMRM) and the Standard Manager.

   PMRM - This manager is designed for professional traders and market makers. Portfolio margin works by aggregating the risk across the entire account, letting the risks of various positions cancel each other out, thereby reducing the overall margin requirements.

   Standard Manager - This manager is designed for the majority of users, catering mostly to non-professional traders or market makers.

2. **Liquidations –** These occur to mitigate the risk of an account becoming insolvent when it falls beneath its margin requirements. If a user falls beneath their margin requirement, their account becomes subject to liquidation. Any user can initiate this process, which may result in the partial or complete liquidation of the account at risk. This is done through an auction process.

3. **Cash –** Cash refers to the designated cash asset, in V2.0, this is USDC. This system facilitates an inherent lending market on the cash asset.

Security Assessment Summary

This review was conducted on the files hosted on the lyra-finance GitHub organisation and were assessed at the following commits:

- `v2-core` : f8c5380
- `lyra-utils` : e2a7042
- `v2-matching` : 0d53aa8

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying issues/vulnerabilities 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 all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

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

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

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 25 issues during this assessment. Categorised by their severity:

- Critical: 4 issues.
- High: 2 issues.
- Medium: 4 issues.
- Low: 6 issues.
- Informational: 9 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Lyra v2 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
LYRA-01	Funds can be Drained from the Protocol by Liquidating an Account During an Asset Transfer	Critical	Resolved
LYRA-02	Bids Can Be Blocked By Sending Option To Liquidator	Critical	Resolved
LYRA-03	Denial-of-Service (DoS) During Manager Health Checks Affects Liquidations	Critical	Resolved
LYRA-04	Liquidator Can Bid With Insufficient Cash	Critical	Resolved
LYRA-05	Account Can Be Created With Arbitrary Manager Address	High	Resolved
LYRA-06	Fully Liquidated Auctions Cannot Be Terminated	High	Resolved
LYRA-07	Assets Can Be Transferred After Auction Started	Medium	Resolved
LYRA-08	Allowances Can Be Frontran	Medium	Closed
LYRA-09	Session Key Deregistration Cooldowns can be Avoided	Medium	Resolved
LYRA-10	Perpetual Asset Funding can be Manipulated by Applying at Peaks and Troughs	Medium	Closed
LYRA-11	Potential DoS Risk In Lyra Protocol Due To Unbounded Loop	Low	Resolved
LYRA-12	Dust Asset Amounts Can Amplify Gas Costs	Low	Resolved
LYRA-13	`AllowList` Can Be Bypassed	Low	Closed
LYRA-14	Allowances Not Cleared On Account Transfer	Low	Closed
LYRA-15	Id Can Overflow In Loop Causing DOS	Low	Resolved
LYRA-16	`ForceWithdraw` Can Be Blocked	Low	Closed
LYRA-17	`ForceWithdraw` Can Cause USDC To Become Stuck	Informational	Resolved
LYRA-18	*Fee-On-Transfer* Tokens Are Not Supported	Informational	Closed
LYRA-19	High Reentrancy Potential in `subAccounts.submitTransfers()`	Informational	Resolved
LYRA-20	Expired Options can be Created	Informational	Closed
LYRA-21	Incorrect Emits	Informational	Resolved
LYRA-22	Maximum Values can be Set Below Current Values	Informational	Closed
LYRA-23	Trading with Self to Make a Loss is Profitable	Informational	Closed
LYRA-24	Oracles Using Perp Market Prices Can Be Manipulated	Informational	Closed
LYRA-25	Miscellaneous General Comments	Informational	Closed

LYRA-01	Funds can be Drained from the Protocol by Liquidating an Account During an Asset Transfer		
Asset	`SubAccounts.sol`, `CashAsset.sol`, `Option.sol`, `DutchAuction.sol`, `BaseManager.sol`, `StandardManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

During an asset transfer which would leave an account insolvent, it is possible to execute a reentrancy attack and liquidate that account before the transfer is fully resolved. This renders the insolvent account solvent and thus means that the transfer goes through when it should not. The insolvent account(s) can be abandoned and the positive recipient of assets can withdraw their fictitious profit.

In the following attack scenario, all accounts (Alice, Bob, Cynthia) are owned by a single attacker.

1. **Alice sets up a transaction which grants profit to Bob and leaves Alice insolvent.** One way of doing this is for Alice to issue Bob with an option which is already expired and which grants Bob significant value, such as a put option at a much higher price than the asset's price at the expiry time. This can be done on an empty account without requiring any assets to be deposited into the protocol.

   However, the effect could also be achieved by transferring out all of the cash asset from an account with a significant existing option liability.

2. **This transaction includes a `managerData` argument which calls an attack contract.** This argument can be supplied to `subAccounts.submitTransfer()`. This function calls `_submitTransfer()` which transfers all assets without any risk check before calling `_managerHook()`, which in turn calls `StandardManager.handleAdjustment()` to perform the risk check. Before the risk check, on line [**266**], this function calls `_processManagerData()` with the `managerData` which Alice submitted. This causes `StandardManager` to call the function `acceptData()` on the attack contract.

3. **The attack contract starts a liquidation auction on Alice.** The attack contract calls `dutchAuction.startAuction()` to start an auction to liquidate Alice. Alice is insolvent, so an insolvent auction is immediately started.

4. **Cynthia bids on Alice's liquidation auction, and fully liquidates her.** The attack contract controls a third account, which we will call Cynthia. Cynthia liquidates Alice by calling `dutchAuction.bid()` for 100% of Alice's account. As we are still within the same single transaction, the liquidation will have a `stepInsolvent` value of zero, and so the full liability of Alice's account will be transferred to Cynthia.

   Critically, however, there is no risk check on Cynthia during this process. All of Alice's liabilities have been transferred to Cynthia before Alice has been risk checked, but Cynthia does not need to pass a risk check to execute a liquidation.

5. **The attack contract returns control flow to `StandardManager`, allowing Alice to pass her risk check.** `StandardManager` now performs a risk check on Alice. As Alice no longer has any assets or liabilities, the risk check quickly passes and the attack transaction ends.

6. **Bob settles his option and withdraws all his cash profit.** Alice is now an empty account. Cynthia is insolvent with no positive assets, but Bob received the profitable side of the option. He can now settle it and withdraw its value in the cash asset.

   Note that none of Alice, Bob or Cynthia has ever deposited any tokens into the protocol.

Recommendations

There are two factors that enable this attack: the reentrancy vulnerability and the ability for liquidators to become insolvent. Whilst fixing either of these would prevent this particular attack, the testing team recommends that both be fixed as they are both significant sources of potential security risk.

1. **Make all the `managerData` external calls as the last step of any transaction, after all checks and effects have been performed.** This significantly reduces reentrancy risk.
2. **Do not allow users to submit `managerData` which causes the protocol to call out to unknown contracts without a reentrancy guard.** Given the large number of contracts within the Lyra protocol, a standard, single contract reentrancy guard might not offer sufficient protection. One approach would be a system wide reentrancy flag which would prevent all calls into the protocol during transactions. Alternatively, consider a whitelist system for the targets of `managerData` and carefully check the whitelisted contracts.
3. **Consider risk checks after liquidation bids are executed.** It is not clear that it is desirable to allow liquidators to make themselves insolvent. As in the case above, this can be used to bypass risk checks and could lead to future security vulnerabilities.

Resolution

The issue has been addressed in pull request #273.

The development team has added the following check to the `BaseManager._processManagerData()` function call, which prevents hijacking program control flow during asset transfer by non-whitelisted managers:

```
if (!whitelistedCallee[managerDatas[i].receiver]) revert BM_UnauthorizedCall()
```

Note, this attack can still be performed by whitelisted accounts. Whitelisted callees are set via `onlyOwner()` function and Lyra appears to manage this function through their business governance processes.

LYRA-02	Bids Can Be Blocked By Sending Option To Liquidator		
Asset	`DutchAuction.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

It is possible to prevent liquidations by frontrunning an `executeBid()` call.

To provide context, liquidations through the `DutchAuction.sol` contract are integral to the proper operation of the Lyra protocol. This is primarily due to their role in executing risk management processes - liquidating high-risk accounts and, in turn, minimizing the risk of the protocol becoming insolvent.

The issue arises due to the requirement that the bidder must only hold the cash asset when performing a bid. This is due to the high gas costs of performing a risk check during liquidation.

However due to this requirement, an attacker could transfer a non-risk adding asset to the bidder right before the `executeBid()` function is called, through frontrunning their transaction. If this occurs, the `executeBid()` function will revert, effectively preventing liquidations from taking place.

The attacker would only need to transfer 1 WEI of an asset to the bidder to perform this attack. Additionally this attack can be performed permissionlessly as the transfer of non-risk adding assets does not require asset allowances.

Recommendations

Consider allowing the bidder to hold other assets when bidding on an auction, and perform a risk check on the bidder once all assets have been transferred.

Resolution

This issue has been addressed in pull request #263.

The development team has added an approval process to option and base assets before an asset can be transferred to a user.

LYRA-03	Denial-of-Service (DoS) During Manager Health Checks Affects Liquidations		
Asset	`SVI.sol`, `LyraVolFeed.sol`, `StandardManager.sol`, `DutchAuction.sol`,		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Configured `SVI` feed bounds can cause denial-of-service (DoS) during Manager health checks, affecting liquidations.

`SVI` library calculates volume based on a chosen strike price. Certain values from configured bounds of `a`, `forwardPrice`, `tao` and `strike` variables will result in reverts in `getVol()` function reading accepted volume data. Reverts in `SVI` will prevent users from bidding in auctions on insolvent accounts, affecting liquidations.

Code comments located in the `SVI.sol` suggest that the following bounds are acceptable:

```
* @param strike desired strike for which to get vol for, in range [0, inf)
* @param a SVI parameter in range (-inf, inf)
* @param b SVI parameter in range [0, inf)
* @param rho SVI parameter in range (-1, 1)
* @param m SVI parameter in range (-inf, inf)
* @param sigma SVI parameter in range (0, inf)
* @param forwardPrice forward price in range [0, inf)
* @param tao time to expiry (in years) in range [0, inf)
```

However, the following exceptions will occur if parameters are set within these expected ranges:

- if `strikePrice` is set to zero, the function will revert with `Overflow`
- if `forwardPrice` exceeds `type(uint56).max`, the function will revert with `Overflow`
- if `forwardPrice` or `tao` are set to zero, the function will revert with `division by modulo 0`
- if `a` is set to a negative non-zero number, the function will revert with `SVI_InvalidParameters`

Since `a`, `forwardPrice` and `tao` variables are set during calls to `LyraVolFeed.acceptData()` where the feed owner has to approve signers to submit data, potential security implications are limited.

However, `strike` variable is controlled by those creating new options. If a user accepts an option with zero value as strike price, it will cause an overflow in all `LyraVolFeed.getVol()` calculations. Subsequently, any call to `StandardManager.getMarginAndMarkToMarket()` will fail as well due to `volFeeds[marketId].getVol()` call on line [**679**].

During bids on insolvent auctions, the `DutchAuction` contract makes a call to `ILiquidatableManager(manager).getMarginAndMarkToMarket(accountId, false, scenarioId)` on line [**676**], which will subsequently revert, preventing auctions on insolvent accounts to complete.

Recommendations

Enforce `forwardPrice`, `tao` and `strike` to only accept non-zero values within their expected range and `a` to be a positive integer (including zero).

It is also recommended to revist all SVI parameters bounds to ensure there are no inconsistencies between expected input into the library parameters and what values options might take on.

Resolution

The issue has been addressed in pull request #12.

The development team has implemented a check to prevent use of zero strike prices and certain bounds on `k` and `w` parameters within the `LyraVolFeed.getVol()`.

The development team has also noted that the main issue surrounds extreme strike price values, as signers misbehaviour is a risk they accept within the overall system behaviour given the signers already hold a high trust assumption.

LYRA-04	Liquidator Can Bid With Insufficient Cash		
Asset	`DutchAuction.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

When `executeBid()` performs `_symmetricManagerAdjustment()` on the cash asset of the bidder, there is no check that the bidder's cash balance remains positive after the adjustment. This could result in a negative cash balance after the bid.

The impact is that an attacker can bid on an auction with an insufficient cash balance, and let their cash balance go negative. The account being liquidated would still get an increase to their cash balance. This is effectively printing cash which could make the protocol insolvent.

Recommendations

The testing team recommends performing risk checks after liquidation bids are executed.

Resolution

This issue has been addressed in pull request #265.

The development team has added the following check to ensure a user has enough cash to bid at an auction:

```
int bidderCashBalance = subAccounts.getBalance(bidderId, cash, 0);
if (bidderCashBalance.toUint256() < cashFromBidder) revert DA_InsufficientCash();
```

LYRA-05	Account Can Be Created With Arbitrary Manager Address		
Asset	`SubAccounts.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Funds can be locked due to allowing `SubAccounts` to be setup with arbitrary manager addresses.

When creating an account through `SubAccounts::createAccount()`, a user can specify an arbitrary address as the manager. This allows the account to receive and hold assets that are not supported by Lyra protocol.

Although this will prevent the SubAccount from interacting normally with legitimate assets in the Lyra protocol, if the manager is later switched to a supported manager, the SubAccount would be able to receive non risk adding assets.

This results in a situation where:

1. Assets in the SubAccount will be stuck as risk checks performed on the account will revert
2. `SubAccount` will no longer be liquidatable as the non-supported asset cannot be transferred
3. `SubAccount` will no longer be able to change their manager

This risk is not only limited to `SubAccounts` created with a malicious manager address. `SubAccounts` which use Portfolio Manager Risk Manager (PMRM) could risk receiving these non-supported assets through a call to `mergeAccounts()`.

A situation where this might occur is where a malicious user sells a `SubAccount` NFT, which holds a non-supported asset, in the open market and the buyer, after purchasing the SubAccount and merging the assets into their legitimate account, end up locking up their legitimate assets.

Recommendations

Implement a whitelist of managers that `SubAccounts` are allowed to use.

Resolution

The original issue had a focus on manager whitelisting, and the development team indicated in pull request #264 that the behaviour of arbitrary manager whitelisting is the intended behaviour.

What is not desired is that an unsupported asset can be accepted as valid after manager changes. The development team has implemented an asset delta check in pull request #264 to all manager transfers, which validates all assets transferring, allowing any manager changes to revert with `SRM_UnsupportedAsset` when the asset is not approved.

It is worth mentioning that it is on the Manager itself to revert when unsupported asset deltas are provided, however, this behaviour appears intentional.

LYRA-06	Fully Liquidated Auctions Cannot Be Terminated		
Asset	`DutchAuction.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

It is possible to lock auctions in a state where it is impossible to terminate or start a new auction.

On line [**482**] in `DutchAuction.sol` there is the condition `if (convertedPercentage > maxOfCurrent)`. If this condition is true then the auction is terminated.

For an auction that can be fully liquidated, `maxOfCurrent` will equal `1e18` and if the liquidator wants to fully liquidate an auction in a single bid, they will input `1e18` when calling `bid()` as the parameter for `percentOfAccount`, this will result in `convertedPercentage` equal to `1e18`.

However in this situation the condition `convertedPercentage > maxOfCurrent` will be `FALSE` which results in the auction not being terminated even though it has been fully liquidated.

In this state if another user attempts to call `bid()` on this auction, the function will revert with `Division or modulo by 0`. Additionally the auction cannot be terminated with `terminateAuction()`, nor can a new auction be started for that SubAccount.

An attacker can potentially use this to perform high risk actions in the Lyra protocol without any prospect of being liquidated. This can significantly increase the risk of Lyra protocol becoming insolvent.

Recommendations

Change the condition to `convertedPercentage >= maxOfCurrent`

Resolution

This issue has been addressed in pull request #270.

The development team has adjusted the code on line [**510**] as per the recommendation, ensuring that fully liquidated auctions can be terminated.

LYRA-07	Assets Can Be Transferred After Auction Started		
Asset	`DutchAuction.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

It is possible to force the bidder to spend more cash than they anticipated through frontrunning.

Accounts are not prevented from transferring assets after the auction has started. This can result in the account being liquidated to frontrun a bid to either transfer assets to another account, which results in the liquidator receiving less assets, or transferring more assets to the account which results in the liquidator spending more cash than they intended.

This poses risks to the bidder as they have no control over how much funds they intend to spend.

Recommendations

Allow the bidder to specify the maximum amount of cash they intend to spend for an auction.

Resolution

The issue has been addressed in pull request #274

The input parameters were amended, as per recommendations, to include a `maxCash` parameter in `DutchAuction.bid()`, to prevent a user from exceeding some unexpected fixed value of cash.

Note, a 'slippage' or unexpected increase in liquidated amount could still occur if `maxCash` is hazardously set by the user to a significant value. As such, it is important the users are aware of what the `maxCash` parameter is aiming to restrict (namely the maximum cash a user wants to spend on a bid) and how to use it effectively.

LYRA-08	Allowances Can Be Frontran		
Asset	`SubAccounts.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`SubAccounts::setAssetAllowances()` and `SubAccounts::setSubIdAllowances()` set allowances directly rather than increasing or decreasing the allowance. This makes it vulnerable to the classic `approve()` frontrunning issue where the allowance can be spent twice through frontrunning when the user is attempting to set a new allowance.

Recommendations

Implement increase and decrease allowance functions.

Resolution

This issue was marked as 'no fix' by the development team, noting:

"(We are) aware of the issue and consider this a reasonable trade off in our use cases."

LYRA-09	Session Key Deregistration Cooldowns can be Avoided		
Asset	`v2-matching/OrderVerifier.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

There is a deregistration cooldown for session keys, but this can be avoided by simply registering the key again, but with an expired date.

The function `deregisterSessionKey()` always gives a grace period of `deregisterSessionKey` to its session key expiries. However, the function `registerSessionKey()` simply allows the session key to be set to any value.

Recommendations

Do not allow the argument `expiry` to `registerSessionKey()` to be less than `block.timestamp + DEREGISTER_KEY_COOLDOWN`.

Resolution

The issue has been addressed in pull request #15.

The development team has adjusted the code on line [39] to only allow updates to expiry that are greater or equal to the current value.

LYRA-10	Perpetual Asset Funding can be Manipulated by Applying at Peaks and Troughs		
Asset	`PerpAsset.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

Because the funding on perpetual assets is calculated by multiplying the current block's funding rate by the entire elapsed time since the last time funding was applied, different total funding rates for the same price movements can be produced by applying funding at different times.

In `PerpAsset._updateFundingRate()`, the current funding rate is calculated and simply multiplied by the time since the last funding rate update. If this occurs at a peak in the funding rate (which can be caused by a sudden shift in asset price or a sudden shift in the impact fee oracles), then this unusually high funding rate is applied to the entire period since the last update.

For perpetual assets which are traded regularly, this is likely to be a low impact issue, as the time since last update would always be small (`_updateFundingRate()` is called on every transfer). However, on a less frequently traded perpetual asset, the impact could be significant.

Recommendations

Consider running automated processes that update perpetual asset funding on a regular basis.

Resolution

This was marked as a non-issue by the development team, noting:

"(We are) aware of the implications of this design and intend to run a regular settler worker to ensure fairness."

LYRA-11	Potential DoS Risk In Lyra Protocol Due To Unbounded Loop		
Asset	`StandardManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The Lyra protocol involves multiple functions that necessitate iterating through all assets to carry out specific calculations or transfers, such as liquidations, risk assessments, and account merging.

At present, the `StandardManager` does not impose any restriction on the quantity of assets that can be whitelisted. This, if increased to a substantially large number, could potentially lead to a Denial of Service (DoS) situation as a result of the gas costs hitting the block's gas limit.

Recommendations

Introduce a cap on the count of assets that can be whitelisted within the `StandardManager`.

Resolution

The issue has been addressed in pull request #271.

The development team has implemented recommended fixes, bounding the number of assets that can be whitelisted within `StandardManager`.

LYRA-12	Dust Asset Amounts Can Amplify Gas Costs		
Asset	`SubAccounts.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

By dusting an account with 1 WEI of assets, an attacker can grief normal users by increasing their gas costs.

Whenever an account engages in a risk-increasing trade, a risk assessment is conducted. This process, which involves gas-intensive computations, requires a loop through all assets currently held by the account.

Potential attackers could exploit this by transferring dust amounts to a target account in a batch. This would significantly increase the transaction cost for the target user due to higher gas requirements.

Assets are also looped through during the liquidation process. An attacker could, in a similar fashion, transfer dust amounts to the account being liquidated by way of frontrunning. This would escalate the gas costs for the liquidator during the liquidation process, potentially altering the incentive structure for liquidation.

Recommendations

The testing team suggests that a minimum transferable amount should be implemented, thereby prevent the transfer of dust amounts.

Resolution

This issue has been addressed in pull request #263.

As with LYRA-02, this issue was also resolved by adding an approval process to option and base assets before an asset can be transferred to a user.

LYRA-13	`AllowList` Can Be Bypassed		
Asset	`AllowList.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Users can bypass `AllowList` by transferring their SubAccount to `Matching`.

The `AllowList` is used to block accounts from trading so they can be liquidated and forced out of the Lyra protocol. This list is updated by valid signers that are part of Lyra Protocol.

However users take advantage of the `Matching` contracts and related modules to continue to interact normally with the Lyra protocol. Given that the blocking is based on account addresses and the necessity of keeping the matching and related modules contracts on the allow list (to avoid disrupting all users whose SubAccounts reside in these contracts), users are able to bypass the intended restrictions.

Recommendations

Due to the matching contracts, the current implementation of allow list does not work as intended. Care must be taken to prevent accidentally removing the matching contracts and modules from the allowlist, which will impact all users using that module.

It is recommended to redesign the allow list system to take into account the issues identified.

Resolution

This was marked as a non-issue, the development team believes that the `AllowList` contract acts as a reference for how the system could be operated and no protocol level checks are required, as a user will be unable to force transactions to go through on the matching contract side.

LYRA-14	Allowances Not Cleared On Account Transfer		
Asset	`SubAccounts.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When asset allowances are configured via the `setAssetAllowances()` function for `subAccounts`, these allowances are not reset during an account Non-Fungible Token (NFT) transfer.

Although allowances are owner specific, this behavior poses potential issues when the account is actively traded in the open market. If an allowance has been previously set and the account is later transferred to another user, the original allowance remains intact even if the account is transferred back to its initial user. This could lead to unexpected behaviors and introduce potential risks for the original user who initially configured the allowance.

Recommendations

Consider resetting asset allowances upon transfer.

Resolution

This issue was marked as 'no fix' by the development team, noting:

"This behaviour was expected, given the complexity of the codebase it would be inefficient to clear allowance (both asset and subID wise) on all assets upon transfer. (...) previously set allowance doesn't affect the new owner when receiving a token, so the security risk is minimized".

LYRA-15	Id Can Overflow In Loop Causing DOS		
Asset	`StandardManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

It is possible for `StandardManager::_arrangePortfolio()` to revert due to `id` overflowing `uint8`.

In StandardManager line [362]: `for (uint8 id = 1; id < 256; id++)`

`id++` will overflow when it reaches the last iteration of the loop when `id` equals 255. One situation where this is possible is when a malicious asset is added to the asset list (refer to issue LYRA-05).

Recommendations

Consider using `uint256` for `id` and `marketId`, which also has the benefits of less gas overhead.

Resolution

This issue has been addressed in pull request #271.

The development team has implemented suggested fixes and are now using `uint` as opposed to `uint8`.

LYRA-16	`ForceWithdraw` Can Be Blocked		
Asset	`CashAsset.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Force withdraw can be blocked since `CashAsset::forceWithdraw()` will withdraw the cash balance to the owner address.

If a user controls an address which was blacklisted by USDC, then they can prevent being force withdrawn by transferring their sub account NFT to the blacklisted address, then `_withdrawCashAmount()` will revert when attempting to transfer USDC to the blacklisted owner address.

The user can at anytime send the NFT to a non blacklisted address if they wish to withdraw normally.

Recommendations

Since an account which only holds cash assets does not pose a risk to the protocol, consider removing the force withdraw functionality.

Resolution

This issue was marked as 'no fix' by the development team, noting:

"Force withdraw is designed for someone who is no longer a legitimate user to leave the system."

As such, the above attack scenario is not a relevant threat.

LYRA-17	`ForceWithdraw` Can Cause USDC To Become Stuck
Asset	`CashAsset.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Force withdraw can cause USDC to become stuck since `CashAsset::forceWithdraw()` withdraws the cash balance to the owner address.

The `SubAccountsManager` contract and the related module contracts handle facilitating trades between subAccounts. Users need to transfer their SubAccount NFT to the `SubAccountsManager` to perform such actions, during which, the SubAccount NFT could be held by the `SubAccountsManager` contract or any of the related module contracts.

If `forceWithdraw()` is called on a SubAccount when the NFT is held by the `SubAccountsManager` contract or module contracts, this will cause USDC to become stuck as there is no method to recover the USDC from these contracts.

Another way USDC can become stuck is if the owner address is held in a user contract which does not support USDC. In normal situations where the user withdraws USDC from `CashAsset.sol`, the user can specify a `recipient` address. However this is not possible during `forceWithdraw()`.

Recommendations

Since an account which only holds cash assets does not pose a risk to the protocol, consider removing the force withdraw functionality.

Alternatively:

1. Ensure the `SubAccountsManager` and related contracts are never removed from the `allowList`, which will prevent `forceWithdraw()` from being called on accounts held in those contracts. See LYRA-13
2. Provide warning in the documentation for users who wish to use a contract to hold SubAccounts, that they must support USDC in the event of `forceWithdraw()`.

Resolution

This issue has been addressed in pull request #19.

The development team has acknowledged that the main impact of this issue is that with some misconfigurations it is possible to `forceWithdraw()` to the `Matching` contract and if the allow list is not set correctly to allow the `Matching` contract, then those funds could not be withdrawn.

As a solution, an `onlyOwner` function was placed to withdraw funds from both the matching and action modules.

LYRA-18	*Fee-On-Transfer* Tokens Are Not Supported
Asset	`WrappedERC20Asset.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Certain tokens are not supported in `WrappedERC20Asset.sol`, for example fee-on-transfer tokens, since the `deposit()` function does not check balance before and after the transfer to ensure the amount received is not reduced.

Recommendations

It is recommended to have a token integration checklist in the documentation to ensure that non-standard tokens are not added. For example: token-interaction-checklist

Resolution

This issue was marked as 'no fix' by the development team as fee-on-transfer tokens are unsupported.

LYRA-19	High Reentrancy Potential in `subAccounts.submitTransfers()`
Asset	`SubAccounts.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The structure of the function `subAccounts.submitTransfers()` is especially vulnerable to reentrancy attacks.

As discussed in LYRA-01, there is a reentrancy risk from the use of `managerData`. Even if this risk were closed, `subAccounts.submitTransfers()` would still create a particular opportunity for reentrancy attacks.

The reason for this higher vulnerability is that the function makes all of the transfers for any number of assets and accounts, and then calls all of the manager hooks for the accounts involved in the order of the submitted transfers. Because `subAccounts` is permissionless, users can create their own managers and assets.

It is possible to submit an `AssetTransfer[]` array consisting of a transfer of user created assets and managers, followed by legitimate Lyra asset transfers. All of the balance adjustments are made first in `subAccounts.submitTransfers()`, and then the user defined managers would be called before any calls were made to Lyra created managers. It is at this point that a reentrancy attack could be executed, similar to LYRA-01.

Note that this applies equally to `permitAndSubmitTransfers()`.

Recommendations

Within `_submitTransfers()`, check that all assets in the transfers have whitelisted all managers in the transfers, and vice versa. Alternatively, simply loop through the transfers and process them individually using `_submitTransfer()`, although this approach would lose the gas benefits of combining the transfers.

Resolution

This issue has been addressed in pull request #279.

The development team has resolved this issue by implementing a reentrancy guard at the `SubAccounts` level. Furthermore, the team aims to conduct follow up investigations on other potential paths of reentrancy throughout the codebase.

LYRA-20	Expired Options can be Created
Asset	`Option.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

It is possible to grant options that have already expired, and so are effectively a sort of disguised cash transfer.

As the testing team understands it, this has no legitimate use within the protocol. It is, however, a tool that might be useful for perpetrating attacks.

Recommendations

The testing team recommends requiring newly created options to have a minimum lifespan before expiry.

Resolution

This issue was marked as 'no fix' by the development team.

LYRA-21	Incorrect Emits
Asset	`CashAsset.sol`, `WrappedERC20Asset.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

`WrappedERC20Asset::withdraw()` emits the wrong event on line [**91**]. The middle value should be `recipient`

```
event Withdraw(uint indexed accountId, address indexed recipient, uint amountAsset);
```

`CashAsset::_withdrawCashAmount()` emits the wrong event on line [**428**].The middle value should be `recipient`

```
event Withdraw(uint accountId, address recipient, uint amountCashBurn, uint wrappedAssetWidrawn);
```

Recommendations

Fix the emitted address as defined by the interface.

Resolution

This issue has been addressed in pull request #271.

The development team has implemented recommended fixes, incorrect events were amended.

LYRA-22	Maximum Values can be Set Below Current Values
Asset	`PMRM.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The admin functions `PMRM.setMaxExpiries()` and `PMRM.setMaxAccountSize()` both allow their configuration values to be set to a level which could be below the values of existing accounts in the system.

This will cause all `PMRM` risk checks for the affected accounts to fail, as both values are checked against the account during `_countExpiriesAndOptions()`, which is called by `_arrangePortfolio()`, which is called by `_assessRisk()`.

To remove this error, the user would need to make a single transaction which takes their account below both of the relevant maximum values.

Recommendations

Be aware of the issue and advise protocol administrators and users appropriately when these values are lowered. Admin should avoid sudden severe drops in these values as users may find it difficult to reconfigure their accounts in such a situation.

Resolution

This issue was marked as 'no fix' by the development, noting:

"(We are) aware of this situation but will keep it as a flexibility to govern the contracts."

LYRA-23	Trading with Self to Make a Loss is Profitable
Asset	`CashAsset.sol`, `DutchAuction.sol`, `BaseManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Because of the socialising of loss within the protocol, it is possible to extract funds from the protocol if a user owns two accounts, and one makes a large and sudden loss to the other, sufficient to make the first account insolvent.

When insolvent accounts are liquidated, the process includes a call to `cashAsset.socializeLoss()`, which will transfer funds from the security module, or else "print" cash, so that the profit making account is able to be fully repaid.

This means that any user who controls two accounts trading with each other can extract cash from the system if they are able to cause one account to incur a significant loss to the other, and go insolvent in the process. The amount by which they go insolvent is the amount of profit that will be extracted.

A general attack flow might be:

1. Wait for the cash balance in the security module to build up.
2. Trade a large WBTC option between two accounts so as to be sitting on both sides of the trade.
3. Secure the call option with WBTC to maximise the loss making potential if WBTC drops in price.
4. Add multiple tiny perps and options (especially when gas is cheap) to increase the cost of liquidation.
5. WBTC changes in price rapidly.
6. One account goes heavily insolvent. Its liquidation will not be economic until auction bids are low and the loss is worse, and so the socialised loss will be higher.
7. So long as the socialised loss is greater than the losing account's securing assets, the attack will be profitable.

Recommendations

The protocol already has measures in place to mitigate this kind of attack: option and perp fees, margin requirements and margin discounting for non-cash assets. The protocol will need to manage its configuration values accordingly, bearing in mind that the socialised loss system provides this additional incentive to intentionally create insolvencies.

Resolution

This issue was marked as 'no fix' by the development team, noting:

"This is the expected behaviour of the system, there is a buffer between users maintence and actually being insolvent."

LYRA-24	Oracles Using Perp Market Prices Can Be Manipulated
Asset	`PerpAsset.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Whilst the full specification of perp oracles is outside the scope of the review, the testing team wishes to add a note on their possible manipulation.

Based on information provided by the development team, the testing team understands that the settlement price of perps will be determined by oracles that use the recent trading prices of such perps on the `matching` contract, (or perhaps in another market).

If the price used by perps is determined by the price on an open market which the users have access to, they could be highly susceptible to short term manipulation. A user could trade with themselves, selling themselves perps far above or below market price. The only penalty for doing this would be the fees, as the funding rate would be paid between the user's own accounts. However, this might allow them to settle a perp held with another user at significantly above or below the spot price of the underlying asset.

This strategy could also be combined with LYRA-23.

Recommendations

Be aware of these concerns in the design and implementation of oracle data and the setting of fee rates.

Resolution

This issue was marked as 'no fix' by the development team, noting:

"Perp prices will be a combination of the spot price and a TWAP of the difference of spot to perp market price. In the case of a sustained TWAP attack, the confidence score would also decrease (this would use a comparison of the long twap to the short twap value used on chain), which would increase margin requirements."

LYRA-25	Miscellaneous General Comments
Asset	`src/*`
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. **safeMint** Consider using `safeMint()` rather than `mint()` for minting SubAccount NFTs.
2. **Code Comments**
   (a) Incorrect comments for the `ISubAccounts::AssetTransfer` struct, `fromAcc` should be debited and `toAcc` should be credited
   (b) The comment on `PMRM.sol` line [478] should say "option", not "perp".
3. **Naming**
   (a) `SubAccounts::permitAndSubmitTransfers()` processes multiple permits, consider renaming to `permitsAndSubmitTransfers()`
   (b) `PRRM::findInArray()` is an internal function. Consider renaming it to `_findInArray()` to follow the convention in the rest of the code.
4. **Inconsistency**
   (a) The documentation for the Security Module state: `Users are not permitted to deposit funds into the SM - only the DAO is able to do so.` However anyone can call `Donate()`
   (b) `OptionEncoding.toSubId()` reverts on an expiry value of zero, but `OptionEncoding.fromSubId()` will return an expiry of zero without error.
   (c) Inconstant use of variable names: `PMRM::handleAdjustment()` uses `riskAdding` while `StandardManager::handleAdjustment()` uses `isRiskReducing`. The testing team recommends keeping these consistent to reduce the risk of errors.
5. **Emits** Consider emitting events for `_setAssetAllowances()` and `_setSubIdAllowances()`
6. **Typo** Spelling mistake `SubAccounts.sol` line [503], `safe on gas` should be `save on gas`
7. **Interface** The following functions are not defined in `IDutchAuction.sol`: `getAuction`, `getMaxProportion`, `getCurrentBidPrice`, `getDiscountPercentage`, `getMarginAndMarkToMarket`, `setBufferMarginPercentage`, `setSolventAuctionParams`, `setInsolventAuctionParams`, `updateScenarioId`, `convertToInsolventAuction`, `terminateAuction`, `bid`, `continueInsolventAuction` and `getAuctionStatus`.
8. **Custom Error** The custom error `OE_StrikeTooLarge` on line [46] of `OptionEncoding.sol` is returning a modified version of the variable `strike`, not the original argument. This might be confusing.
9. **Zero strike** `OptionEncoding` supports a strike price of zero. Consider whether it might be desirable to disallow this.

10. **Nonce** Users currently lack the ability to cancel orders, having to wait instead for their expiration, regardless of market conditions. To enhance flexibility, consider introducing an incremental nonce, allowing users to invalidate all orders simultaneously by calling `incrementNonce()`
11. **Gas Optimisation** There are many instances in the contracts where the array length is not cached in a for loop. Consider caching all array lengths to save gas.

Recommendations

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

Resolution

The comments above have been acknowledged by the development team, and relevant changes actioned in pull request #271, where applicable.

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 `foundry` framework was used to perform these tests and the output is given below.

```
    Running 1 test for test/v2-matching/Matching.t.sol:TestMatching
[PASS] testVerifyAndMatch() (gas: 55805)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.60ms

Running 2 tests for test/v2-core/feeds/BaseFeed.t.sol:TestBaseLyraSpotFeed
[PASS] testCanUseMultipleSigners() (gas: 178252)
[PASS] testCannotUpdateWithNoSigners() (gas: 32991)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 3.83ms

Running 16 tests for test/v2-core/assets/PerpAsset.t.sol:TestPerpAsset
[PASS] testGetUnsettledAndUnrealizedCash() (gas: 185056)
[PASS] testPerpApplyFundingOnAccount() (gas: 162211)
[PASS] testPerpConstructor() (gas: 1918944)
[PASS] testPerpGetFundingRate() (gas: 83609)
[PASS] testPerpGetImpactPrices() (gas: 33993)
[PASS] testPerpGetIndexPrice() (gas: 15220)
[PASS] testPerpGetPerpPrice() (gas: 23203)
[PASS] testPerpHandleAdjustment() (gas: 171446)
[PASS] testPerpHandleManagerChange() (gas: 5206624)
[PASS] testPerpRealizePNLWithMark() (gas: 122617)
[PASS] testPerpSetImpactFeeds() (gas: 37119)
[PASS] testPerpSetPerpFeed() (gas: 24556)
[PASS] testPerpSetSpotFeed() (gas: 24519)
[PASS] testPerpSetStaticInterestRate() (gas: 22147)
[PASS] testPerpSettleRealizedPNLAndFunding() (gas: 122632)
[PASS] testPerpSettleRealizedPNLAndFundingWrongCaller() (gas: 16201)
Test result: ok. 16 passed; 0 failed; 0 skipped; finished in 8.00ms

Running 1 test for test/v2-core/Allowances.t.sol:TestAllowances
[PASS] testAllowancesOnTransfer() (gas: 661234)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.72ms

Running 9 tests for test/v2-core/liquidation/DutchAuction.t.sol:TestDutchAuction
[PASS] testContinueInsolventAuction() (gas: 589411)
[PASS] testForceStartAuction() (gas: 588361)
[PASS] testLiquidatePartial() (gas: 1352291)
[PASS] testSetBufferMarginPercentage() (gas: 21442)
[PASS] testSetInsolventAuctionParams() (gas: 21620)
[PASS] testSetSolventAuctionParams() (gas: 32148)
[PASS] testStartAuction() (gas: 316703)
[PASS] testTerminateAuction() (gas: 318391)
[PASS] testUpdateScenarioId() (gas: 466593)
Test result: ok. 9 passed; 0 failed; 0 skipped; finished in 13.81ms

Running 3 tests for test/v2-core/risk-managers/PMRM.t.sol:TestStandardManager
[PASS] testPMRMMerge() (gas: 438095)
[PASS] testPMRM_perpTransfer() (gas: 401993)
[PASS] testSetFeeds() (gas: 63124)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 7.76ms

Running 17 tests for test/v2-core/assets/CashAsset.t.sol:TestCashAsset
[PASS] testCashAccrueInterest() (gas: 272467)
[PASS] testCashConstructor() (gas: 12888)
[PASS] testCashDeposit() (gas: 250945)
[PASS] testCashDepositToNewAccount() (gas: 299796)
[PASS] testCashDisableWithdrawFee() (gas: 267209)
[PASS] testCashForceWithdraw() (gas: 260893)
[PASS] testCashHandleAdjustment() (gas: 56216)
[PASS] testCashHandleManagerChange() (gas: 5104499)
[PASS] testCashSetInterestRateModel() (gas: 29575)
[PASS] testCashSetLiquidationModule() (gas: 20367)
```

```
[PASS] testCashSetSmFee() (gas: 20599)
[PASS] testCashSetSmFeeRecipient() (gas: 19701)
[PASS] testCashSocializeLoss() (gas: 246399)
[PASS] testCashWithdraw() (gas: 313298)
[PASS] testCashWithdrawAll() (gas: 274884)
[PASS] testGetCashToStableExchangeRate() (gas: 17405)
[PASS] testUpdateSettledCash() (gas: 64993)
Test result: ok. 17 passed; 0 failed; 0 skipped; finished in 19.44ms

Running 5 tests for test/v2-core/assets/InterestRateModel.t.sol:TestInterestRateModel
[PASS] testIRMConstructor() (gas: 554371)
[PASS] testIRMGetBorrowInterestFactor(uint256,uint256) (runs: 256)
[PASS] testIRMGetBorrowInterestFactorNoTime() (gas: 9047)
[PASS] testIRMGetBorrowRate() (gas: 155696)
[PASS] testIRMGetUtilRate() (gas: 41061)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 32.29ms

Running 2 tests for test/v2-core/risk-managers/StandardManager.t.sol:TestStandardManager
[PASS] testMergeAccountsMergeSelf() (gas: 37978)
[PASS] testWithdrawWrappedAssetMarginCheck() (gas: 319574)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 6.13ms

Running 4 tests for test/v2-core/libraries/AssetDeltaLib.t.sol:TestAssetDeltaLib
[PASS] testADLAddToAssetDeltaArray() (gas: 26304)
[PASS] testADLAddToAssetDeltaArrayTooLong() (gas: 2232662)
[PASS] testADLGetDeltasFromArrayCache() (gas: 23386)
[PASS] testADLGetDeltasFromSingleAdjustment() (gas: 3634)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 8.24ms

Running 1 test for test/v2-core/periphery/PerpSettlementHelper.t.sol:TestPerpSettlementHelper
[PASS] testPSHAcceptData() (gas: 597115)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.04ms

Running 7 tests for test/v2-core/assets/OptionAsset.t.sol:TestOptionAsset
[PASS] testGetOptionDetailsErrors() (gas: 15720)
[PASS] testGetOptionDetailsFuzz(uint256,uint256,bool) (runs: 256)
[PASS] testOptionCalcSettlementValue() (gas: 56357)
[PASS] testOptionConstructor() (gas: 9873)
[PASS] testOptionGetSettlementValue() (gas: 11796)
[PASS] testOptionHandleAdjustment() (gas: 156734)
[PASS] testOptionHandleManagerChange() (gas: 5804645)
Test result: ok. 7 passed; 0 failed; 0 skipped; finished in 31.80ms

Running 4 tests for test/v2-core/feeds/AllowList.t.sol:TestAllowList
[PASS] testAcceptData(uint64,bytes,uint256,uint64,address) (runs: 256)
[PASS] testAddSigner(address,bool) (runs: 256)
[PASS] testSetAllowListEnabled(uint64,address) (runs: 256)
[PASS] testSetHeartbeat(uint64) (runs: 256)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 255.30ms

Running 2 tests for test/v2-core/feeds/LyraSpotDiffFeed.t.sol:TestLyraSpotDiffFeed
[PASS] testFuzzInDataAndUpdateSpotDiffFeed(uint64,int96,uint64) (runs: 256)
[PASS] testSetSpotFeed() (gas: 139555)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 151.46ms

Running 1 test for test/v2-core/periphery/OptionSettlementHelper.t.sol:TestOptionSettlementHelper
[PASS] testOSHAcceptData() (gas: 510513)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.28ms

Running 4 tests for test/v2-core/SecurityModule.t.sol:TestDutchAuction
[PASS] testDonate() (gas: 201934)
[PASS] testRequestPayout() (gas: 515427)
[PASS] testWhitelistModule() (gas: 32716)
[PASS] testWithdraw() (gas: 258300)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 13.32ms

Running 21 tests for test/v2-core/SubAccounts.t.sol:TestSubAccounts
[PASS] testAddHeldAsset() (gas: 296878)
[PASS] testAssetAdjustment() (gas: 138058)
[PASS] testAssetHook() (gas: 45032)
```

```
[PASS] testChangeManager() (gas: 5146000)
[PASS] testCreateAccount() (gas: 81846)
[PASS] testCreateAccountWithApproval() (gas: 106560)
[PASS] testGetAccountBalances() (gas: 34115)
[PASS] testGetBalance() (gas: 18501)
[PASS] testGetUniqueAssets() (gas: 33673)
[PASS] testManagerAdjustment() (gas: 84405)
[PASS] testManagerHook() (gas: 598031)
[PASS] testPermit() (gas: 199240)
[PASS] testPermitAndSubmitTransfer() (gas: 586650)
[PASS] testPermitAndSubmitTransfers() (gas: 805325)
[PASS] testRemoveHeldAsset() (gas: 254959)
[PASS] testSetAssetAllowances() (gas: 87461)
[PASS] testSetSubIdAllowances() (gas: 71752)
[PASS] testSubmitTransfer() (gas: 377194)
[PASS] testSubmitTransfer_allowances() (gas: 337434)
[PASS] testSubmitTransfer_assetAllowance() (gas: 344416)
[PASS] testSubmitTransfers() (gas: 477448)
Test result: ok. 21 passed; 0 failed; 0 skipped; finished in 20.96ms

Running 5 tests for test/v2-core/assets/WrappedERC20Asset.t.sol:TestWrappedERC20Asset
[PASS] testWERC20Deposit() (gas: 241686)
[PASS] testWERC20HandleAdjustment() (gas: 104647)
[PASS] testWERC20HandleManagerChange() (gas: 5194576)
[PASS] testWERC20Withdraw() (gas: 270376)
[PASS] testWERC20WithdrawErrors() (gas: 290142)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 6.54ms

Running 3 tests for test/v2-matching/SubAccountsManager-test.t.sol:TestSubAccountsManager
[PASS] testCreateSubAccount() (gas: 347981)
[PASS] testDepositSubAccount() (gas: 104796)
[PASS] testSubAccountWithdraw() (gas: 123580)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 28.61ms

Running 5 tests for test/v2-core/feeds/LyraForwardFeed.t.sol:TestLyraForwardFeed
[PASS] testCannotUpdateFwdFeedAfterDeadline() (gas: 41295)
[PASS] testCannotUpdateFwdFeedFromInvalidSigner() (gas: 45623)
[PASS] testForwardFeedSetSettlementHeartBeat(uint64) (runs: 256)
[PASS] testFuzzInDataAndUpdateFwdFeed(uint64,uint256,uint256,int96,uint64) (runs: 256)
[PASS] testRevertsWhenFetchingInvalidExpiry(uint64) (runs: 256)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 275.31ms

Running 22 tests for test/pocs/POC.t.sol:TestPOC
[PASS] testFailAuctionDOSBidMaliciousOption_vuln() (gas: 7275133)
[PASS] testFailCannotTerminateAuction_vuln() (gas: 429139)
[PASS] testFailGetMarginMalOption() (gas: 7275222)
[PASS] testFailLiquidateMidTransfer() (gas: 1918465)
[PASS] testFailMaliciousAuctionDOSBidFrontrun_vuln() (gas: 1264668)
[PASS] testFailMaliciousOptionPrintCash() (gas: 7275156)
[PASS] testFailMaliciousOptionStartAuction_vuln() (gas: 7275133)
[PASS] testFailMergeAccountMaliciousOption() (gas: 7275178)
[PASS] testFailMergeNegativeCashAccounts() (gas: 480447)
[PASS] testFailMergeNegativeCashAccounts2() (gas: 478794)
[PASS] testFailTransferMaliciousOption() (gas: 7275221)
[PASS] testFailWinAuctionNegativeCash_vuln() (gas: 470051)
[PASS] testFailWithdrawCashMalOption() (gas: 7275156)
[PASS] testFundingDiff1() (gas: 200679)
[PASS] testFundingDiff2() (gas: 174058)
[PASS] testSelfTransferOption() (gas: 635856)
[PASS] testSocializeSocializedLoss() (gas: 260705)
[PASS] testSubmitTransfer_tiny() (gas: 217192)
[PASS] testTransferAssetAfterStartAuction_vuln() (gas: 646345)
[FAIL. Reason: Overflow() Counterexample:
      ↪  calldata=0xfed34955000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
      ↪  args=[0, 0, 0, 0, 0, 1, 1, 0, 0]]
      ↪  testTransferZeroStrikeOption(int128,uint128,int56,int64,uint64,uint56,uint64,uint64,uint64) (runs: 0)
[PASS] testUSDCStuck() (gas: 318051)
[PASS] testZeroCashTransfer() (gas: 223818)
Test result: FAILED. 21 passed; 1 failed; 0 skipped; finished in 507.65ms
```

```
Running 3 tests for test/v2-core/feeds/LyraRateFeed.t.sol:TestLyraRateFeed
[PASS] testCanAddSigner() (gas: 32656)
[PASS] testCanPassInDataAndUpdateRateFeed() (gas: 60170)
[PASS] testFuzzInDataAndUpdateRateFeed(uint64,int96,uint64) (runs: 256)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 256.87ms

Running 3 tests for test/v2-core/feeds/LyraVolFeed.t.sol:TestLyraVolFeed
[PASS] testCannotPassInTimestampHigherThanExpiry(uint64) (runs: 256)
[FAIL. Reason: Overflow() Counterexample:
      ↪  calldata=0xf6300dd1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
      ↪  args=[0, 0, 0, 0, 0, 0, 1000000000000000001 [1e18], 1, 0, 1]]
      ↪  testFuzzInDataAndUpdateVolFeed(uint256,int128,uint128,int56,int64,uint64,uint256,uint64,uint256,uint128) (runs: 27)
[PASS] testRevertsWhenFetchingInvalidExpiry() (gas: 25862)
Test result: FAILED. 2 passed; 1 failed; 0 skipped; finished in 177.15ms

Running 1 test for test/v2-core/feeds/LyraSpotFeed.t.sol:TestLyraSpotFeed
[PASS] testFuzzInDataAndUpdateSpotFeed(uint64,uint96,uint64) (runs: 256)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 180.00ms

Running 2 tests for test/v2-core/libraries/PermitAllowanceLib.t.sol:TestPermitAllowanceLib
[PASS] testPALHash() (gas: 3062)
[PASS] testPALHashFuzz(address,uint256,uint256,uint256,(address,uint256,uint256)[],(address,uint256,uint256,uint256)[]) (runs: 256)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 535.62ms
Ran 25 test suites: 142 tests passed, 2 failed, 0 skipped (144 total tests)

Failing tests:
Encountered 1 failing test in test/pocs/POC.t.sol:TestPOC
[FAIL. Reason: Overflow() Counterexample:
      ↪  calldata=0xfed34955000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
      ↪  args=[0, 0, 0, 0, 0, 1, 1, 0, 0]]
      ↪  testTransferZeroStrikeOption(int128,uint128,int56,int64,uint64,uint56,uint64,uint64,uint64) (runs: 0)

Encountered 1 failing test in test/v2-core/feeds/LyraVolFeed.t.sol:TestLyraVolFeed
[FAIL. Reason: Overflow() Counterexample:
      ↪  calldata=0xf6300dd1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
      ↪  args=[0, 0, 0, 0, 0, 0, 1000000000000000001 [1e18], 1, 0, 1]]
      ↪  testFuzzInDataAndUpdateVolFeed(uint256,int128,uint128,int56,int64,uint64,uint256,uint64,uint256,uint128) (runs: 27)
```

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.

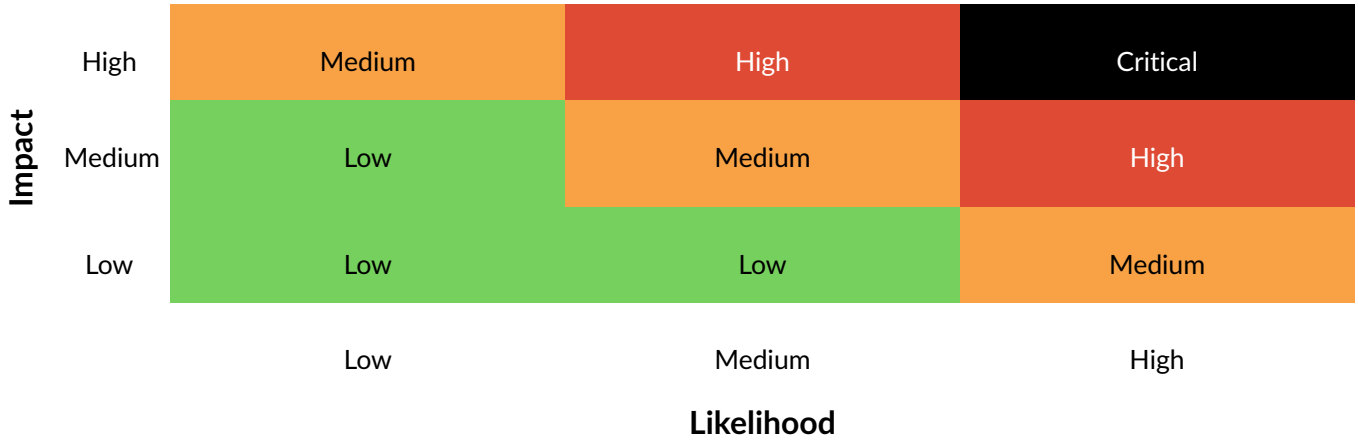

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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].