Security Review Report
NM-0099 Libre

(Jul 07, 2023)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 4

4 System Overview 5

5 Risk Rating Methodology 7

6 Issues 8
- 6.1 [Critical] USDC is permanently locked in the SubscriptionBook . . . 8
- 6.2 [Critical] _instantSettlement(...) does not implement fee deduction . . . 9
- 6.3 [High] Admin cancellation of an order may lead to an insufficient security token balance in the RedemptionBook . . . 10
- 6.4 [High] Admin cancellation of the first order in the queue will lead to failure of settling orders . . . 10
- 6.5 [High] Missing zero check for investorId could be abused to drain tokens . . . 12
- 6.6 [High] order.amountPaid is updated with the amount sent to the treasury instead of beneficiary . . . 13
- 6.7 [Medium] Enabling a module without resetting the registries leads to incorrect behavior of static checks . . . 14
- 6.8 [Medium] Investors and dealers can add any wallet even if it does not belong to them . . . 15
- 6.9 [Medium] Investors are unable to redeem the entire amount . . . 15
- 6.10 [Medium] Last operation timestamp in InvestorRegistry can be overridden . . . 16
- 6.11 [Medium] The function settleOrders(...) never marks order as done . . . 16
- 6.12 [Medium] The last item is never checked due to an incorrect implementation of the function getValueAfterTimestamp(...) . . . 17
- 6.13 [Medium] investorBalance[] is not updated when investors add or remove a wallet . . . 17
- 6.14 [Medium] The function rebalanceSettlements(...) lacks global state update like the function _partialSettlement(...) . . . 18
- 6.15 [Medium] settledAmount is never reset in _instantSettlement(...) . . . 18
- 6.16 [Low] Functions initialize(...) can be frontrun . . . 19
- 6.17 [Low] Incorrect definition of INITIAL_RESTRICTED_PERIOD_ALLOWANCE constant variable . . . 20
- 6.18 [Low] Malicious contract manager can continue to pass hasAccess after manager role is changed . . . 20
- 6.19 [Info] Duplicate elements in operationModules[] could cause deleteModule(...) to work incorrectly . . . 21
- 6.20 [Info] Redundant loop check in the function _editModuleClone(...) . . . 21
- 6.21 [Info] Registering the same module again will not produce the desired outcome . . . 22
- 6.22 [Info] The total storage variable in IdSorter can be manipulated . . . 22
- 6.23 [Info] Typo in string used to derive storage key . . . 23
- 6.24 [Info] insertId(...) fails to insert an element equal to the end . . . 23
- 6.25 [Best Practice] Duplicated logic between the HoldingsModule and the BidsAggregationLimitModule . . . 24
- 6.26 [Best Practice] Emitting address instead of index of the wallet . . . 24
- 6.27 [Best Practice] Improve efficiency of insertId(...) by utilizing a hint for faster position finding . . . 25
- 6.28 [Best Practice] Including zero check for denominator when performing division . . . 25
- 6.29 [Best Practice] Missing event emission . . . 25
- 6.30 [Best Practice] Prevent initialization of implementation contracts . . . 26
- 6.31 [Best Practice] Redundant investorBalance getter in SecurityToken contract . . . 26
- 6.32 [Best Practice] Unnecessary parent before and after token transfer function call . . . 26
- 6.33 [Best Practice] Unnecessary setting of IS_INSTRUMENT and FUND_ID in InstrumentRegistry . . . 27
- 6.34 [Best Practice] Unused storage variables . . . 27
- 6.35 [Best Practice] Verify whether the transfer was successful and the token balance after transfer using order amount . . . 27
- 6.36 [Best Practice] checkCorrectInvestor(...) could receive investorId instead of _orderId . . . 28
- 6.37 [Best Practice] updateInvestorLastOperationTimestamp(...) parameter can be investorId . . . 29

7 Documentation Evaluation 29

8 Test Suite Evaluation 30
- 8.1 Tests Output . . . 30
- 8.2 Code Coverage . . . 31
- 8.3 Slither . . . 31

9 About Nethermind 32

1 Executive Summary

This document outlines the security review conducted by Nethermind for the `Libre` protocol. `Libre`, also known as Libra, is a general-purpose asset tokenization platform. Its primary purpose is to provide individuals and organizations with access to funds at lower costs and with lower investment minimums. The platform aims to democratize access to financial assets by leveraging blockchain technology and smart contracts.

The audited code comprises 4535 lines of Solidity. The `Libre` team has provided detailed documentation explaining the protocol summary, the flow of the subscription book and the redemption book, and the interaction between each instrument with registries and rule engines. The audit was conducted in three separate commits. The initial commit hash is e22cc7, followed by the second commit 6625b3, and the final review commit is e88db8.

The audit was performed using: (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contracts, and (d) creation of test cases. **Along this document, we report** 37 points of attention, where 2 are classified as `Critical`, 4 are classified as `High`, 9 are classified as `Medium`, 3 are classified as `Low`, and 19 are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

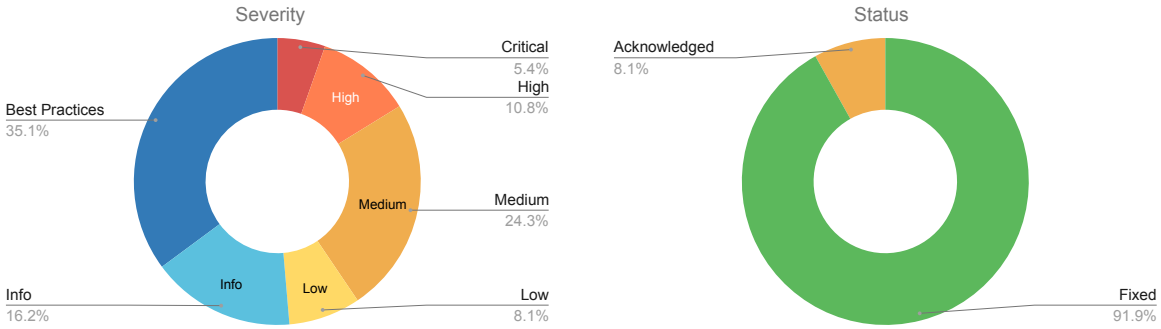

Fig 1: Distribution of issues: Critical (2), **High** (4), **Medium** (9), **Low** (3), **Undetermined** (0), **Informational** (6), **Best Practices** (13). **Distribution of status: Fixed** (34), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	June 19, 2023
Response from Client	June 30, 2023
Final Report	July 7, 2023
Methods	Manual Review, Automated Analysis
Repository	https://github.com/NethermindEth/libre-platform-contracts/tree/
Commit Hash (Initial Audit)	e88db8da563a0acbe3bffa7af911714215759d4c
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	SecurityToken.sol	105	10	9.5%	23	138
2	IdSorter.sol	106	37	34.9%	11	154
3	RedemptionBook.sol	655	100	15.3%	130	885
4	EternalRegistryStorage.sol	193	9	4.7%	35	237
5	DealerRulesEngine.sol	162	14	8.6%	29	205
6	SystemDeployment.sol	58	13	22.4%	7	78
7	PermissionedContract.sol	14	16	114.3%	5	35
8	RulesEngine.sol	253	13	5.1%	43	309
9	OperationModule.sol	20	9	45.0%	3	32
10	SubscriptionBook.sol	389	41	10.5%	102	532
11	registries/DealerRegistry.sol	33	7	21.2%	13	53
12	registries/InstrumentRegistry.sol	277	8	2.9%	39	324
13	registries/InvestorRegistry.sol	244	30	12.3%	40	314
14	registries/BaseRegistry.sol	127	83	65.4%	30	240
15	registries/FundRegistry.sol	20	6	30.0%	6	32
16	registries/RoleRegistry.sol	75	4	5.3%	12	91
17	registries/JurisdictionRegistry.sol	23	12	52.2%	8	43
18	registries/BaseUserRegistry.sol	52	9	17.3%	16	77
19	lib/TimeOperations.sol	86	33	38.4%	9	128
20	lib/MonotonicQueue.sol	113	68	60.2%	17	198
21	lib/BitMask.sol	23	41	178.3%	6	70
22	modules/general/HoldingsModule.sol	252	8	3.2%	37	297
23	modules/subscriptions/BidsAggregationLimitModule.sol	117	5	4.3%	13	135
24	modules/subscriptions/SubscriptionSizeModule.sol	158	4	2.5%	26	188
25	modules/allowlist/AllowlistModuleDealerExample1.sol	119	15	12.6%	20	154
26	modules/allowlist/AllowlistModuleLibreExample1.sol	149	19	12.8%	24	192
27	modules/allowlist/AllowlistModuleInstrumentExample1.sol	57	6	10.5%	14	77
28	interfaces/IRedemptionBook.sol	40	39	97.5%	6	85
29	interfaces/IDealerRegistry.sol	13	35	269.2%	7	55
30	interfaces/ISubscriptionBook.sol	34	28	82.4%	5	67
31	interfaces/ISecurityToken.sol	21	42	200.0%	8	71
32	interfaces/IEternalRegistryStorage.sol	29	118	406.9%	16	163
33	interfaces/IRoleRegistry.sol	17	70	411.8%	12	99
34	interfaces/IInvestorRegistry.sol	47	134	285.1%	25	206
35	interfaces/IFundRegistry.sol	8	19	237.5%	4	31
36	interfaces/IOrderPipeline.sol	33	96	290.9%	19	148
37	interfaces/IUserRegistry.sol	10	32	320.0%	7	49
38	interfaces/IDealerRulesEngine.sol	29	77	265.5%	12	118
39	interfaces/IInstrumentRegistry.sol	43	74	172.1%	11	128
40	interfaces/IRulesEngine.sol	74	51	68.9%	8	133
41	interfaces/operations/IAdvisedLockBidCheck.sol	6	9	150.0%	1	16
42	interfaces/operations/ILockRedemptionCheck.sol	4	8	200.0%	1	13
43	interfaces/operations/ICreateRedemptionCheck.sol	10	11	110.0%	1	22
44	interfaces/operations/IAdvisedCreateBidCheck.sol	11	12	109.1%	1	24
45	interfaces/operations/IBidConfirmationCheck.sol	6	10	166.7%	1	17
46	interfaces/operations/IAdvisedRedemptionConfirmationCheck.sol	10	11	110.0%	1	22
47	interfaces/operations/IAdvisedBidCancelationCheck.sol	6	9	150.0%	1	16
48	interfaces/operations/IForcedRedemptionCheck.sol	6	9	150.0%	1	16
49	interfaces/operations/IOptionalFlagsInstrumentCheck.sol	8	23	287.5%	3	34
50	interfaces/operations/IOptionalFlagsDealerCheck.sol	10	25	250.0%	3	38
51	interfaces/operations/ISMFillCheck.sol	6	9	150.0%	1	16
52	interfaces/operations/IAdvisedRedemptionCancelationCheck.sol	6	9	150.0%	1	16
53	interfaces/operations/IForcedTransferCheck.sol	6	10	166.7%	1	17
54	interfaces/operations/ISendCheck.sol	6	10	166.7%	1	17
55	interfaces/operations/IReceiveCheck.sol	6	10	166.7%	1	17
56	interfaces/operations/ILibreFlagsCheck.sol	10	22	220.0%	3	35

	Contract	LoC	Comments	Ratio	Blank	Total
57	interfaces/operations/IAdvisedLockRedemptionCheck.sol	6	9	150.0%	1	16
58	interfaces/operations/ISettleBidsCheck.sol	12	13	108.3%	1	26
59	interfaces/operations/ISettleRedemptionsCheck.sol	10	11	110.0%	1	22
60	interfaces/operations/ITransferCheck.sol	6	10	166.7%	1	17
61	interfaces/operations/ISMOrderCheck.sol	6	10	166.7%	1	17
62	interfaces/operations/IAdvisedBidConfirmationCheck.sol	10	11	110.0%	1	22
63	interfaces/operations/ICreateBidCheck.sol	10	11	110.0%	1	22
64	interfaces/operations/ISMTradeCheck.sol	6	11	183.3%	1	18
65	interfaces/operations/IAdvisedCreateRedemptionCheck.sol	11	12	109.1%	1	24
66	interfaces/operations/IRedemptionConfirmationCheck.sol	9	10	111.1%	1	20
67	interfaces/operations/ILockBidCheck.sol	4	8	200.0%	1	13
68	interfaces/operations/IOperationModule.sol	50	14	28.0%	4	68
	Total	**4535**	**1762**	**38.9%**	**895**	**7192**

3 Summary of Issues

	Finding	Severity	Update
1	`USDC` is permanently locked in the `SubscriptionBook`	Critical	Fixed
2	`_instantSettlement(...)` does not implement fee deduction	Critical	Fixed
3	Admin cancellation of an order may lead to an insufficient security token balance in the RedemptionBook	High	Fixed
4	Admin cancellation of first order in queue will lead to failing of settling orders	High	Fixed
5	Missing zero check for `investorId` could be abused to drain tokens	High	Fixed
6	`order.amountPaid` is updated with the amount sent to the treasury instead of beneficiary	High	Fixed
7	Enabling a module without resetting the registries leads to incorrect behaviour of static checks	Medium	Fixed
8	Investors and dealers can add any wallet even if it does not belong to them	Medium	Fixed
9	Investors are unable to redeem the entire amount	Medium	Fixed
10	Last operation timestamp in InvestorRegistry can be overridden	Medium	Fixed
11	The function `settleOrders(...)` never marks order as done	Medium	Fixed
12	The last item is never checked due to an incorrect implementation of the function `getValueAfterTimestamp(...)`	Medium	Fixed
13	`investorBalance[]` is not updated when investors add or remove a wallet	Medium	Fixed
14	`rebalanceSettlements(...)` lacks global state update like `_partialSettlement(...)`	Medium	Fixed
15	`settledAmount` is never reset in `_instantSettlement(...)`	Medium	Fixed
16	Functions `initialize(...)` can be frontrun	Low	Fixed
17	Incorrect definition of `INITIAL_RESTRICTED_PERIOD_ALLOWANCE` constant variable	Low	Fixed
18	Malicious contract manager can continue to pass `hasAccess` after manager role is changed	Low	Fixed
19	Duplicate elements in `operationModules[]` could cause `deleteModule(...)` to work incorrectly	Info	Fixed
20	Redundant loop check in the function `_editModuleClone(...)`	Info	Fixed
21	Registering the same module again will not produce the desired outcome	Info	Fixed
22	The `total` storage variable in `IdSorter` can be manipulated	Info	Fixed
23	Typo in string used to derive storage key	Info	Fixed
24	`insertId(...)` fails to insert an element equal to the end	Info	Fixed
25	Duplicated logic between HoldingsModule and BidsAggregationLimitModule	Best Practices	Acknowledged
26	Emitting address instead of index of the wallet	Best Practices	Acknowledged
27	Improve efficiency of `insertId(...)` by utilizing a hint for faster position finding	Best Practices	Fixed
28	Including zero check for denominator when performing division	Best Practices	Fixed
29	Missing event emission	Best Practices	Fixed
30	Prevent initialization of implementation contracts	Best Practices	Fixed
31	Redundant `investorBalance` getter in `SecurityToken` contract	Best Practices	Fixed
32	Unnecessary parent before and after token transfer function call	Best Practices	Acknowledged
33	Unnecessary setting of `IS_INSTRUMENT` and `FUND_ID` in `InstrumentRegistry`	Best Practices	Fixed
34	Unused storage variables	Best Practices	Fixed
35	Verify whether the transfer was successful and the token balance after transfer using order amount	Best Practices	Fixed
36	`checkCorrectInvestor(...)` could receive `investorId` instead of `_orderId`	Best Practices	Fixed
37	`updateInvestorLastOperationTimestamp(...)` parameter can be `investorId`	Best Practices	Fixed

4 System Overview

The main contracts of the system included:

a) **SystemDeployment**
b) **SubscriptionBook**
c) **RedemptionBook**
d) **SecurityToken**
e) **RulesEngine**
f) **InvestorRegistry**
g) **DealerRegistry**
h) **FundRegistry**
i) **JurisdictionRegistry**
j) **InstrumentRegistry**
k) **RoleRegistry**
l) **Modules**

Fig. 2 presents the interaction diagram of contracts.

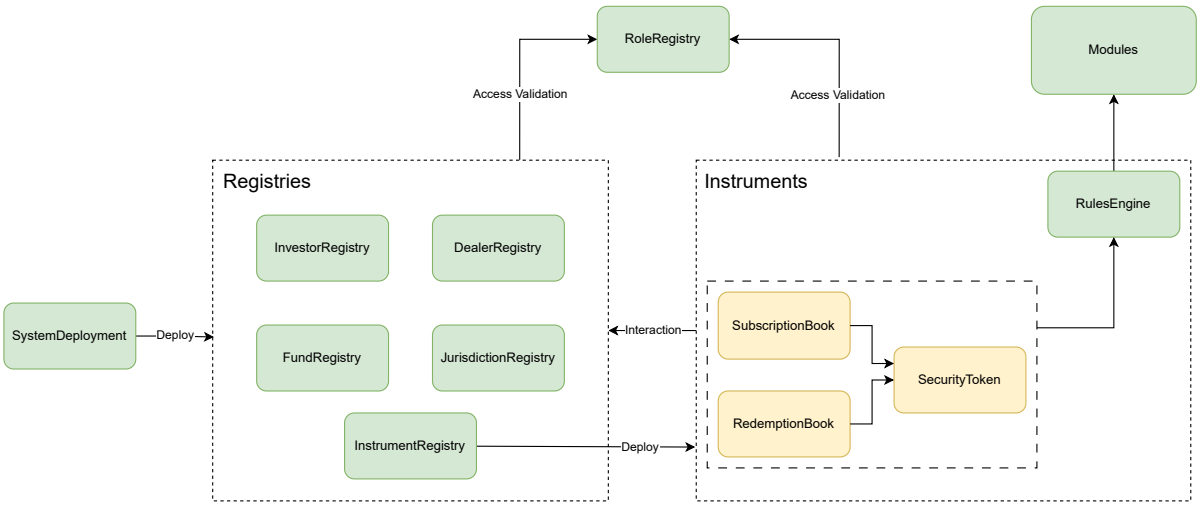

Fig. 2: Interaction Diagram of Contracts

The **SystemDeployment** contract is responsible for deploying all registries and initializing them. The deploying registries include **RoleRegistry**, **InvestorRegistry**, **DealerRegistry**, **FundRegistry**, **JurisdictionRegistry** and **InstrumentRegistry** .

The **SubscriptionBook** contract is responsible for implementing the logic and functionality related to the subscription process within the Libre platform. It manages the order book for subscriptions and handles the creation, confirmation, locking, and settlement of subscription orders.

The **RedemptionBook** contract is responsible for implementing the logic and functionality related to the redemption process within the Libre platform. It manages the order book for redemptions and handles the creation, confirmation, locking, and settlement of redemption orders.

The **SecurityToken** contract is responsible for managing and ensuring compliance with the rules associated with an asset by utilizing the **RulesEngine** . Each security token represents an asset on the platform. Also, it can track an investor's overall balance across multiple wallets.

The **RulesEngine** contract is responsible for handling and validating compliance rules associated with various participants, assets, and functionalities within the platform.

The **InvestorRegistry** contract is designed to store and manage the information of all investors participating in the platform. The contract may interact with other contracts and components within the platform to validate investor information and ensure compliance.

The **DealerRegistry** contract, similarly to InvestorRegistry, is designed to store and manage the information of all dealers or entities representing and onboarding investors to the platform.

The **FundRegistry** contract is responsible for storing and managing information related to funds or entities that issue assets on the platform. The contract would ensure that the necessary compliance rules are applied to each fund and that the associated assets and instruments are properly configured.

The **JurisdictionRegistry** contract is designed to store and manage information associated with jurisdictions and compliance rules specific to those jurisdictions.

The **InstrumentRegistry** contract is responsible for registering and managing instruments. Instruments represent various financial assets or investment products issued and traded on the platform. Each instrument can have multiple components, such as a security token, a rules engine, a subscription book, and a redemption book.

The **RoleRegistry** contract is responsible for managing and tracking roles associated with different participants in the system. It is designed to handle access control and authorization within the platform by assigning and revoking roles to different addresses or entities. The contract maintains a hierarchical tree structure of roles, where each role can have parent roles and grant or revoke sub-roles.

The **Modules** contracts are components or plugins that can be added to different contracts within the Libre platform to extend functionalities or implement specific business logic. Modules can be registered or attached to the respective contracts through the **RulesEngine** to enable their functionality.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] USDC is permanently locked in the SubscriptionBook

File(s): SubscriptionBook.sol/L352, SubscriptionBook.sol/L429

Description: The SubscriptionBook allows users to lock their USDC to receive security tokens. However, once the USDC is deposited into the contract and the orders are settled, it becomes impossible to withdraw these funds from the contract. As a result, the USDC is locked permanently, with no option for withdrawal. As described below, the code comments on this functionality as a TODO task.

```
function settleOrders(uint256 _lastOrderId, uint256 _percentageToSettle, bytes32 _role) external override {
    ...
    uint256 orderId = allOrders[i];
    Order memory order = orders[orderId];

    uint256 amountSettled = (order.amount * _percentageToSettle) / 10000;

    if (amountSettled > totalToSettle) {
        amountSettled = totalToSettle;
    }
    uint256 amountToIssue = (amountSettled * 10 ** securityToken.decimals()) / navPerShare;

    securityToken.issue(order.beneficiary, amountToIssue);
    // @audit There is a TODO here to implement the withdraw function
    // TODO: Add logic for the fund to receive the payment tokens

    totalToSettle -= amountSettled;
    _partialSettlement(orderId, order.investorId, amountSettled);
...
}
```

Similarly, the logic for the fund should be implemented in `_instantSettlement(...)`, presented below:

```
function _instantSettlement(uint256 _orderId) internal {
    Order memory order = orders[_orderId];

    uint256 auditedNav = instrumentRegistry.getAuditedNavPerShare(instrumentId);
    uint256 amountToIssue = (order.amount * 10 ** securityToken.decimals()) / auditedNav;
    securityToken.issue(order.beneficiary, amountToIssue);

    // @audit The logic for the fund to receive the payment tokens should also be implemented here
    _partialSettlement(_orderId, order.investorId, order.amount);
    _decreaseInvestorOrders(order.investorId);
   ...
}
```

Recommendation(s): Ensure implementing a function for the fund to receive the payment tokens in `_instantSettlement(...)` and `settleOrders(...)`, or another withdrawal mechanism.

Status: Fixed

Update from the client: Fixed in e88db8d

6.2 [Critical] `_instantSettlement(...)` does not implement fee deduction

File(s): RedemptionBook.sol

Description: `_instantSettlement(...)` is automatically triggered to calculate the instant settlement of an order when all four phases are executed in the same transaction. The function is presented below:

```
function _instantSettlement(uint256 _orderId) internal {
    Order memory order = orders[_orderId];

    // TODO: Type of NAV used to TBD with the client
    uint256 unauditedNavPerShare = instrumentRegistry.getUnauditedNavPerShare(instrumentId);
    uint256 amountToPay = order.amount * unauditedNavPerShare / 10 ** usdc.decimals();

    securityToken.burn(order.amount);
    usdc.transfer(order.beneficiary, amountToPay);

    // TODO: Deduct fee

    // Update storage variables affected by the settlement
    _partialSettlement(_orderId, order.investorId, 0, order.amount);
    _decreaseRedeemersOrders(order.investorId);
    orders[_orderId].confirmed = false; // mark order as done

    removeId(_orderId);
}
```

The function `settleOrders(...)` implements the same functionality, but it calculates the settlement of **a set** of orders. However, in `_instantSettlement(...)`, there is no fee deduction implementation. The same is applied to the function `rebalanceSettlements(...)`.

```
function rebalanceSettlements(uint256 _lastOrderToRebalance, bytes32 _role) external {
    _checkRoleHasAccess(_role);
    uint256 auditedNav = instrumentRegistry.getAuditedNavPerShare(instrumentId);

    uint256 decimals = 10 ** securityToken.decimals();
   ...
}
```

Recommendation(s): Ensure `_instantSettlement(...)` and `rebalanceSettlements(...)` implements the same fee deduction process as implemented in `settleOrders(...)`.

Status: Fixed

Update from the client: Fixed in e88db8d

6.3 [High] Admin cancellation of an order may lead to an insufficient security token balance in the RedemptionBook

File(s): RedemptionBook.sol

Description: In the RedemptionBook contract, the admin can cancel any existing order by calling the function `adminCancelOrder(...)`. When a locked order is canceled, the corresponding amount of security tokens are transferred back to the beneficiary in the function `_afterCancelOrderCheck(...)`.

```
function _afterCancelOrderCheck(uint256 _orderId, Order memory _order, uint256 _submitter) internal {
    // update the investors count
    _decreaseRedeemersOrders(_order.investorId);

    if (_order.confirmed) {
        confirmedAmount -= _order.amount;
        confirmedAmountPerInvestor[_order.investorId] -= _order.amount;

        // If locked update the locked amount
        ////////////////////////////////////////////
        // @audit Not consider _order.amountSettled
        ////////////////////////////////////////////
        if (!_order.available) {
            amountLocked -= _order.amount;
            // @audit should be _order.amount - _order.amountSettled instead
            securityToken.transfer(_order.beneficiary, _order.amount);
            removeId(_orderId);
        }
    }

    // Reduce the aggregated amount of orders received
    aggregatedAmount -= _order.amount;

    // delete the order
    delete orders[_orderId];
    emit OrderCanceled(_orderId, _submitter);
}
```

However, there is a potential issue where an order could be partially settled when the admin attempts to cancel it. However, the `amountSettled` is not deducted when sending the `securityToken` back to the beneficiary. Consequently, a user could receive the total amount of security tokens while their order has already been partially settled, meaning they have also received USDC previously. Additionally, since security tokens are burned upon settlement, the contract may have insufficient security tokens available for other users to cancel or settle their orders.

Recommendation(s): Consider the `amountSettled` when the admin cancels an order and transfers the security tokens back to the investor.

Status: Fixed

Update from the client: Fixed in e88db8d

6.4 [High] Admin cancellation of the first order in the queue will lead to failure of settling orders

File(s): SubscriptionBook.sol

Description: Admin has the privilege of canceling orders even after the tokens are locked for a particular order. In that case, the orderId is removed from the queue. But in a special case, after an order is canceled, the settle order functionality fails.

```
function settleOrders(uint256 _lastOrderId, uint256 _percentageToSettle, bytes32 _role) external override {
    ...
    if (_lastOrderId < nextOrderToSettle) {
        revert IOrderPipelineOrderAlreadySettledInCurrentRound();
    }

    uint256[] memory allOrders = getIdsOrderFrom(nextOrderToSettle == 0 ? start : nextOrderToSettle, _lastOrderId);
    ...
    nextOrderToSettle = idsQueue[_lastOrderId].next;
}
```

Let's take a scenario where there are five orders(1,2,3,4,5)

1. Admin first settles orders till 2. Queue is updated and `nextOrderToSettle` is updated to 3;
2. Later, the admin cancels order 3, which updates the `start` variable in `idSorter` contract to 4, and the order queue looks like (1,2,4,5);
3. Then the admin wants to settle orders till 5, which should settle orders 4 and 5. But since `nextOrderToSettle` is not updated, `settleOrders` function will fetch for orders from 3 to 5, which reverts because of `IdSorterToParamNotFound()` error;

```
function testSettleOrders() public {
    uint256 _amount = 1000;
    instrumentRegistry.updateAuditedNavPerShare(
        keccak256("LIBRE_FUND_ADMIN_ROLE"), subscriptionBook.instrumentId(), 1000
    );
    dealerRegistry.allowDealer(0x0, LIBRE_DEALER, LIBRE_INSTRUMENT, true);
    investorRegistry.allowInvestor(LIBRE_DEALER_ROLE, LIBRE_INVESTOR);
    paymentToken.mint(LIBRE_INVESTOR_WALLET, _amount); // 1000
    dealerRegistry.allowDealer(0x0, LIBRE_DEALER, LIBRE_INSTRUMENT, true);
    investorRegistry.allowInvestor(LIBRE_DEALER_ROLE, LIBRE_INVESTOR_2);
    paymentToken.mint(LIBRE_INVESTOR_WALLET_2, _amount + 1000); // 2000
    dealerRegistry.allowDealer(0x0, LIBRE_DEALER, LIBRE_INSTRUMENT, true);
    investorRegistry.allowInvestor(LIBRE_DEALER_ROLE, LIBRE_INVESTOR_3);
    paymentToken.mint(LIBRE_INVESTOR_WALLET_3, _amount + 100); // 1100
    dealerRegistry.allowDealer(0x0, LIBRE_DEALER, LIBRE_INSTRUMENT, true);
    investorRegistry.allowInvestor(LIBRE_DEALER_ROLE, LIBRE_INVESTOR_4);
    paymentToken.mint(LIBRE_INVESTOR_WALLET_4, _amount + 200); // 1200
    dealerRegistry.allowDealer(0x0, LIBRE_DEALER, LIBRE_INSTRUMENT, true);
    investorRegistry.allowInvestor(LIBRE_DEALER_ROLE, LIBRE_INVESTOR_5);
    paymentToken.mint(LIBRE_INVESTOR_WALLET_5, _amount - 100); // 900

    // five investors create order, confirm and lock tokens
    vm.startPrank(LIBRE_INVESTOR_WALLET);
    uint256 orderId1 = subscriptionBook.investorCreateOrder(_amount);
    subscriptionBook.investorConfirmOrder(orderId1);
    paymentToken.approve(address(subscriptionBook), type(uint256).max);
    subscriptionBook.investorLockTokens(orderId1);
    vm.stopPrank();

    vm.startPrank(LIBRE_INVESTOR_WALLET_2);
    uint256 orderId2 = subscriptionBook.investorCreateOrder(_amount + 1000);
    subscriptionBook.investorConfirmOrder(orderId2);
    paymentToken.approve(address(subscriptionBook), type(uint256).max);
    subscriptionBook.investorLockTokens(orderId2);
    vm.stopPrank();

    vm.startPrank(LIBRE_INVESTOR_WALLET_3);
    uint256 orderId3 = subscriptionBook.investorCreateOrder(_amount + 100);
    subscriptionBook.investorConfirmOrder(orderId3);
    paymentToken.approve(address(subscriptionBook), type(uint256).max);
    subscriptionBook.investorLockTokens(orderId3);
    vm.stopPrank();

    vm.startPrank(LIBRE_INVESTOR_WALLET_4);
    uint256 orderId4 = subscriptionBook.investorCreateOrder(_amount + 200);
    subscriptionBook.investorConfirmOrder(orderId4);
    paymentToken.approve(address(subscriptionBook), type(uint256).max);
    subscriptionBook.investorLockTokens(orderId4);
    vm.stopPrank();

    vm.startPrank(LIBRE_INVESTOR_WALLET_5);
    uint256 orderId5 = subscriptionBook.investorCreateOrder(_amount - 100);
    subscriptionBook.investorConfirmOrder(orderId5);
    paymentToken.approve(address(subscriptionBook), type(uint256).max);
    subscriptionBook.investorLockTokens(orderId5);
    vm.stopPrank();

    subscriptionBook.settleOrders(orderId2, 10000, keccak256("LIBRE_FUND_ADMIN_ROLE")); // admin settling orders till 2
    subscriptionBook.adminCancelOrder(orderId3, keccak256("LIBRE_FUND_ADMIN_ROLE")); // admin cancelling order 3
    subscriptionBook.settleOrders(orderId5, 10000, keccak256("LIBRE_FUND_ADMIN_ROLE")); // admin trying to settle
↪   orders till 5 but reverts
}
```

While canceling the order nextOrderToSettle is not updated.

```
function _afterCancelOrderCheck(uint256 _orderId, Order memory _order, uint256 _submitter) internal {
    ...

    if (_order.confirmed) {
        confirmedAmount -= _order.amount;
        confirmedAmountPerInvestor[_order.investorId] -= _order.amount;

        // If locked update the locked amount
        if (!_order.available) {
            amountLocked -= _order.amount;
            paymentToken.transfer(_order.beneficiary, _order.amount);
            removeId(_orderId);
        }
    }

    ...

    // @audit update the nextOrderToSettle

    // delete the order
    delete orders[_orderId];
    emit OrderCanceled(_orderId, _submitter);
}
```

Recommendation(s): Consider updating the `nextOrderToSettle` variable while the admin is canceling the order to resolve this issue in case the canceled order equals to `nextOrderToSettle`.

```
if (cancelled_id == nextOrderToSettle) then set nextOrderToSettle;
```

Status: Fixed

Update from the client: Fixed in b502050

6.5 [High] Missing zero check for `investorId` could be abused to drain tokens

File(s): SubscriptionBook.sol

Description: In the function `checkCorrectInvestor(...)`, it is called `investorRegistry` to query the `investorId` of the sender. Similarly, in the function `checkCorrectDealer(...)`, `dealerRegistry` queries the `dealerId` and `investorRegistry` to query the `dealerId` of the given investor. However, all these external calls lack zero checks, which means if there is not a valid investor or dealer, these registries will return `bytes32(0)` instead of reverting.

```
////////////////////////////////////////////////////
// @audit both dealerID and investorId could be 0
////////////////////////////////////////////////////
function checkCorrectDealer(bytes32 _investorId) internal view {
    bytes32 dealerId = dealerRegistry.getIdFromWallet(msg.sender);
    if (investorRegistry.getDealer(_investorId) != dealerId) {
        revert IOrderPipelineUnauthorized();
    }
}
```

As a result, an attacker can create orders for `investorId = bytes32(0)` since it passed all the checks in the functions `investorCreateOrder(...)` or `checkCorrectInvestor(...)`. The problem is in the function `_afterLockOrderCheck(...)`, the dealer can specify the wallet that will pay the USDC `_investorWallet`

```
////////////////////////////////////////////////////////////////////
// @audit can be used to drain anyone approved token to the contract
////////////////////////////////////////////////////////////////////
function _afterLockOrderCheck(uint256 _orderId, Order memory _order, address _investorWallet, uint256 _submitter)
    internal
{
    address sender;
    if (_investorWallet == address(0x0)) {
        sender = msg.sender;
    } else {
        if (_order.investorId != investorRegistry.getIdFromWallet(_investorWallet)) {
            revert IOrderPipelineWrongInvestor();
        }
        sender = _investorWallet;
    }

    usdc.transferFrom(sender, address(this), _order.amount);
```

As we can see, in case `investorId = bytes32(0)`, any wallet that did not belong to any investor will be valid to pay since it checked `_order.investorId != investorRegistry.getIdFromWallet(_investorWallet)`. For instance, if an investor previously added a wallet, approved the contract to transfer USDC, then removed the wallet, this wallet can be drained if he forgot to revoke the allowance.

Recommendation(s): Add zero checks for `investorId` and `dealerId`.

Status: Fixed

Update from the client: Fixed in e88db8d

6.6 [High] `order.amountPaid` is updated with the amount sent to the treasury instead of beneficiary

File(s): `RedemptionBook.sol`

Description: In the function `settleOrders(...)`, when `amountSettled` is greater than zero, the `_calculateFeeDeduction(...)` is called to compute the `amountToInvestor` and `amountToPlatform` to be transferred to the beneficiary and treasury, respectively. The function is listed below:

```
function settleOrders(uint256 _lastOrderId, uint256 _percentageToSettle, bytes32 _role) public override {
  ...
  if (amountSettled != 0) {
      _registerInvestorSettlement(order.investorId, amountSettled);
      (uint256 amountToInvestor, uint256 amountToPlatform) =
          _calculateFeeDeduction(order.investorId, amountSettled, periodStart);

      // Burn ST and transfer payment to beneficiary

      securityToken.burn(amountSettled);
      // do payment and update storage
      uint256 amountToPay = amountToInvestor * unauditedNav / 10 ** securityToken.decimals();
      usdc.transfer(order.beneficiary, amountToPay);
      if (amountToPlatform != 0) {
          amountToPay = amountToPlatform * unauditedNav / 10 ** securityToken.decimals();
          usdc.transfer(treasury, amountToPay);
      }

      totalToSettle -= amountSettled;
      /*
       @audit the amountToPay passed to _partialSettlement is the amount in some scenarios
              is calculated based on the amountToInvestor and others on the amountToPlatform
      */
      _partialSettlement(orderId, order.investorId, amountToPay, amountSettled);
  }
  ...
}
```

However, when `amountToPlatform` is greater than zero, the amount to be transferred to the treasury is calculated by overwriting the variable `amountToPay`. Consequently, the paid amount passed to `_partialSettlement(...)` for updating the `orders[_orderId].amountPaid` contains the amount transferred to the treasury, not the beneficiary.

```
function _partialSettlement(uint256 _orderId, bytes32 _investorId, uint256 _amountPaid, uint256 _amountSettled)
    internal
{
    // Update order state
    orders[_orderId].amountPaid += _amountPaid;
    orders[_orderId].amountSettled += _amountSettled;
   ...
}
```

Consequences: The `order.amountPaid` can contain the amount sent to the treasury and not the real amount sent to the beneficiary. The beneficiary can receive a greater amount than required.

```
function rebalanceSettlements(uint256 _lastOrderToRebalance, bytes32 _role) external {
    ...
    bool loopToLast = true;
    while (loopToLast) {
        Order memory order = orders[start];
        uint256 totalToBePaid = order.amount * auditedNav / decimals;
        /*
            @audit order.amountPaid can contain the amount sent to the treasury
                   and not the real amount sent to the beneficiary.
        */
        if (totalToBePaid > order.amountPaid) {
            usdc.transfer(order.beneficiary, totalToBePaid - order.amountPaid);
            orders[start].amountPaid = totalToBePaid;
        }
    ...
}
```

Recommendation(s): Ensure that the `orders[_orderId].amountPaid` is updated with the paid amount to the beneficiary instead of the registry.

Status: Fixed

Update from the client: Fixed in e88db8d

6.7 [Medium] Enabling a module without resetting the registries leads to incorrect behavior of static checks

File(s): RulesEngine.sol, DealerRulesEngine.sol

Description: In both rules engines, certain registries are reset when a new module is registered to ensure that static checks are performed accurately. However, the problem arises when enabling or disabling a module, as the registries are not reset accordingly.

Consider the following scenario:

1. Module A is added, and the registries are reset to trigger static checks;
2. Module A is subsequently disabled, and Module B is added. The registries are reset once again, and static checks are now performed exclusively for Module B;
3. If Module A is enabled again, the registries are not reset, and the static checks continue to be applied only for Module B. This results in bypassing all the rules defined in module A;

Recommendation(s): Consider resetting the registries when enabling a module.

Status: Fixed

Update from the client: Fixed in ee9ca15

6.8 [Medium] Investors and dealers can add any wallet even if it does not belong to them

File(s): InvestorRegistry.sol

Description: The InvestorRegistry contract currently allows investors to add an unlimited number of wallets, even if those wallets do not belong to them. The only condition is that the wallet has not been added by someone else at the same moment. However, this assumption is insufficient to ensure that the wallet address belongs to the investor.

```
function addWallet(address _wallet) external {
    bytes32 hashedWallet = keccak256(abi.encodePacked(_wallet));
    if (getBytes(hashedWallet, OWNED_BY) != 0x0) {
        revert InvestorRegistryWalletIsClaimedBySomeoneElse();
    }
    bytes32 hashedCurrentWallet = keccak256(abi.encodePacked(msg.sender));
    bytes32 investorId = getBytes(hashedCurrentWallet, OWNED_BY);
    if (!getBool(investorId, IS_INVESTOR)) {
        revert InvestorRegistryNotAnInvestor();
    }
    _pushAddressArray(investorId, OWNED_WALLETS, _wallet);
    _setBytes(hashedWallet, OWNED_BY, investorId);
    emit WalletAdded(_wallet);
}
```

As a result, an attacker can add any wallet address they know belongs to other investors. If, by mistake, the other investor approves USDC to the contract, the attacker can steal it.

Recommendation(s): Consider adding a mechanism to verify the ownership of a wallet, such as implementing a signature scheme or requiring the wallet to confirm on-chain after being added by an investor.

Status: Fixed

Update from the client: Fixed in e88db8d

6.9 [Medium] Investors are unable to redeem the entire amount

File(s): HoldingsModule.sol, BidsAggregationLimitModule.sol, SubscriptionSizeModule.sol

Description: Currently, in these modules, the redemption check returns false if the net holding of an investor after redemption is smaller than the minimum holding per investor. However, it fails to consider the scenario where investors want to redeem the full amount. In such cases, the net holding of the investor after the redemption action becomes 0, causing the check to fail. The code snippet below demonstrates the check implemented in the function _checkRedemptionConfirmation(...):

```
// @audit does not allow to redeem the full amount, _incomingRedemption = investorNetHolding
if (investorNetHolding - _incomingRedemption < minHoldingPerInvestor) {
    return (false, "SubscriptionSizeModule: Incoming amount breaches min investor holding");
} else {
    return (true, "");
}
```

Recommendation(s): Consider modifying the check to allow investors to redeem the full amount.

Status: Fixed

Update from the client: Fixed in c76db59

6.10 [Medium] Last operation timestamp in InvestorRegistry can be overridden

File(s): InvestorRegistry.sol

Description: The `InvestorRegistry` contract inherits the `EternalRegistryStorage` contract, which allows for setting value for any key in some mappings. Certain special keys need to be reserved for special variables to make it work. However, the last operation timestamp in the code snippet below is a hash of `LAST_OPERATION_TIMESTAMP` and `_instrumentId`, and has not been reserved. As a result, any sender with access to the function `setUint(...)` can override its value.

```
function updateInvestorLastOperationTimestamp(bytes32 _investorId, bytes32 _instrumentId, uint256 _lastTimestamp)
    external
    onlyInstrument(_instrumentId)
{
    // TODO: If restrictions are added for certain keys, this one should be secured when adding an instrument
    // @audit key is hash of LAST_OPERATION_TIMESTAMP and _instrumentId, thus is not reserved
    _setUint(_investorId, keccak256(abi.encodePacked(LAST_OPERATION_TIMESTAMP, _instrumentId)), _lastTimestamp);
}

// @audit In BaseRegistry, sender that has access can override the last operation timestamp
function setUint(bytes32 _id, bytes32 _key, uint256 _value, bytes32 _senderRole) external {
    _checkHasAccess(_id, _senderRole);
    _onlyConfigurableKey(_id, _key);
    _setUint(_id, _key, _value);
    _afterEntryUpdate(_id);
}
```

Recommendation(s): Consider reserving the key for the last operation timestamp in the `InvestorRegistry` contract to prevent it from being overridden by the function `setUint(...)`.

Status: Fixed

Update from the client: Fixed in e88db8d

6.11 [Medium] The function `settleOrders(...)` never marks order as done

File(s): SubscriptionBook.sol

Description: The function `settleOrders(...)` is used to settle a set of orders after the three phases are executed (create order, confirm the order, and lock tokens). When the settlement updates the state variables, `orders[orderId].amount` can be decreased to zero. In this case, the order should be marked as done. However, `orders[_orderId].confirmed` remains `true`.

```
function settleOrders(uint256 _lastOrderId, uint256 _percentageToSettle, bytes32 _role) external override {
    ...
    if (orders[orderId].amount == 0) {
      lastEmptyOrder = orderId;
      amountOfEmptyOrders++;
      _decreaseInvestorOrders(order.investorId);
  }
  ...
}
```

Recommendation(s): Ensure that orders are marked as done by adding the code line:

```
if (orders[orderId].amount == 0) {
lastEmptyOrder = orderId;
amountOfEmptyOrders++;
_decreaseInvestorOrders(order.investorId);
+ orders[_orderId].confirmed = false;
}
```

Status: Fixed

Update from the client: Fixed in e88db8d

6.12 [Medium] The last item is never checked due to an incorrect implementation of the function `getValueAfterTimestamp(...)`

File(s): `MonotonicQueue.sol`

Description: The function `getValueAfterTimestamp(...)` currently utilizes a binary search to locate the first item in the queue with a greater or equal timestamp. However, there is an issue with the initial range setup for the binary search. The range is set as `low = head` and `high = tail - 1`, but the while loop condition `low < high` causes the last item in the queue to be excluded from comparison with the query parameter. If the queue only contains one element, the while loop will not execute, and the function will return the `low` item without validating its timestamp.

```
uint256 low = head;
uint256 high = tail - 1; // @audit element at tail - 1 is never checked
while (low < high) {
    uint256 mid = (low + high) / 2;
    if (_q.data[mid].timestamp < _timestamp) {
        low = mid + 1;
    } else {
        high = mid;
    }
}

if (low >= tail) {
    return (0, 0);
}

return (_q.data[low].timestamp, _q.data[low].value);
```

Recommendation(s): Consider fixing the issue by changing the initial range of the binary search.

Status: Fixed

Update from the client: Fixed in e88db8d

6.13 [Medium] `investorBalance[]` is not updated when investors add or remove a wallet

File(s): `BaseUserRegistry.sol`

Description: Each investor can own multiple wallets in the Libre platform. Each wallet will have its balance of security tokens, and the total amount of security tokens an investor owns should be stored in the `investorBalance[]` array in the `InvestorRegistry`. However, it has been observed that this `investorBalance[]` array is not updated when investors add or remove their wallets. As a result, there is a discrepancy between the actual total balance and the balance recorded in the `InvestorRegistry`.

```
////////////////////////////////////////////////////////
// @audit Not update investorBalance[] in InvestorRegistry
////////////////////////////////////////////////////////
function removeWallet(uint256 _index) external override {
    bytes32 _userId = ownedBy[msg.sender];
    _checkIsValidId(_userId);

    uint256 length = ownedWallets[_userId].length;
    address walletToDelete = ownedWallets[_userId][_index];
    ownedWallets[_userId][_index] = ownedWallets[_userId][length - 1];
    ownedWallets[_userId].pop();

    delete ownedBy[walletToDelete];

    emit WalletRemoved(_index);
}
```

In addition, the total balance of an investor is used when the admin wants to call the function `adminForceRedemption()`. This means that an investor can block the admin from executing a forced redemption by simply removing any wallet with a non-zero balance.

Recommendation(s): Consider updating `investorBalance[]` in `InvestorRegistry` when adding or removing a wallet.

Status: Fixed

Update from the client: Fixed in 5fa9623. Wallets removal functionality deleted. Given that wallets can be only added once to the system, they will belong to a unique investor. Before being added, wallets can't hold any of the assets issued in the platform, meaning that, at additional time, they are empty, so no balance update is needed.

6.14 [Medium] The function `rebalanceSettlements(...)` lacks global state update like the function `_partialSettlement(...)`

File(s): RedemptionBook.sol

Description: In the `RedemptionBook` contract, the `rebalanceSettlements(...)` function is responsible for rebalancing the settlements of orders in the current period using the audited NAV (Net Asset Value). If the amount to be paid to an investor is higher than the paid amount, the contract pays the remaining amount and updates the data in the `Order` struct. However, it fails to update global state variables, such as `settledAmount` and `confirmedAmount`, which are typically updated in the normal flow when using the `settleOrders(...)` function.

```
function rebalanceSettlements(uint256 _lastOrderToRebalance, bytes32 _role) external {
    _checkRoleHasAccess(_role);
    uint256 auditedNav = instrumentRegistry.getAuditedNavPerShare(instrumentId);

    uint256 decimals = 10 ** securityToken.decimals();

    bool loopToLast = true;
    while (loopToLast) {
        Order memory order = orders[start];
        uint256 totalToBePaid = order.amount * auditedNav / decimals;
        if (totalToBePaid > order.amountPaid) {
            usdc.transfer(order.beneficiary, totalToBePaid - order.amountPaid);
            orders[start].amountPaid = totalToBePaid;
        }
        ////////////////////////////////////////////////////////////////////////////
        // @audit amountSettled of Order is updated but not the global state settledAmount
        ////////////////////////////////////////////////////////////////////////////
        orders[start].amountSettled = order.amount;

        if (start == _lastOrderToRebalance) {
            loopToLast = false;
        }

        emit OrderRebalanced(start);
        _decreaseRedeemersOrders(order.investorId);
        orders[start].confirmed = false; // mark order as done
        removeId(start);
    }
}
```

Recommendation(s): Consider updating the global state variables to maintain consistency in behavior when using both the `settleOrders(...)` and `rebalanceSettlements(...)` functions.

Status: Fixed

Update from the client: Fixed in e88db8d

6.15 [Medium] `settledAmount` is never reset in `_instantSettlement(...)`

File(s): SubscriptionBook.sol, RedemptionBook.sol

Description: The function `settleOrders(...)` gets the `currentPeriod` to check if the round changed (`forPeriod != currentPeriod`). When `currentPeriod` is different to `forPeriod`, the `settledAmount` is reset, as described below:

```
function settleOrders(uint256 _lastOrderId, uint256 _percentageToSettle, bytes32 _role) external override {
    ...
    (uint256 currentPeriod,,) = instrumentRegistry.currentRedemptionPeriod(instrumentId);

    if (forPeriod != currentPeriod) {
        forPeriod = currentPeriod;
        settledAmount = 0;
        nextOrderToSettle = 0;
    }
    ...
}
```

However, `_instantSettlement(...)` does not check if the `currentPeriod` changed to reset `settledAmount`.

```
function _instantSettlement(uint256 _orderId) internal {
    Order memory order = orders[_orderId];

    uint256 auditedNav = instrumentRegistry.getAuditedNavPerShare(instrumentId);
    uint256 amountToIssue = (order.amount * 10 ** securityToken.decimals()) / auditedNav;

    securityToken.issue(order.beneficiary, amountToIssue);

    _partialSettlement(_orderId, order.investorId, order.amount);
    _decreaseInvestorOrders(order.investorId);
    orders[_orderId].confirmed = false;

    removeId(_orderId);
}
```

Recommendation(s): Ensure `_instantSettlement(...)` resets `settledAmount` when the round changes in both contracts `SubscriptionBook` and `RedemptionBook`.

Status: Fixed

Update from the client: Fixed in e88db8d

6.16 [Low] Functions `initialize(...)` can be frontrun

File(s): src/

Description: The function `initialize(...)` is responsible for initializing important contract states, but anyone can call it. An attacker could potentially initialize the contract before the legitimate deployer does, hoping the victim continues to use the compromised contract. In the best case for the victim, they would notice the compromise and have to redeploy their contract, incurring additional gas costs. The code snippet below shows an example of the function `initialize(...)` in the `RulesEngine` contract:

```
function initialize(
    bytes32 _instrumentId,
    IInvestorRegistry _investorRegistry,
    IDealerRegistry _dealerRegistry,
    IFundRegistry _fundRegistry,
    IInstrumentRegistry _instrumentRegistry,
    IRoleRegistry _roleRegistry,
    IEternalRegistryStorage _jurisdictionRegistry
) external initializer {
    instrumentId = _instrumentId;
    investorRegistry = _investorRegistry;
    fundRegistry = _fundRegistry;
    dealerRegistry = _dealerRegistry;
    instrumentRegistry = _instrumentRegistry;
    roleRegistry = _roleRegistry;
    jurisdictionRegistry = _jurisdictionRegistry;
}
```

Recommendation(s): Consider using the constructor to initialize non-proxied contracts or employ a factory contract for deploying and initializing contracts in an atomic transaction.

Status: Fixed

Update from the client: Fixed in e88db8d

6.17 [Low] Incorrect definition of INITIAL_RESTRICTED_PERIOD_ALLOWANCE constant variable

File(s): RedemptionBook.sol

Description: Incorrect definition of constant variables can lead to critical issues like reading wrong values from storage.

In RedemptionBook contract INITIAL_RESTRICTED_PERIOD_ALLOWANCE is incorrectly defined with the same value of INITIAL_SUBSCRIPTION_RESTRICTED_PERIOD_ALLOWANCE. This wrong uint value is read to the allowedPercent variable in _calculateFeeDeduction function.

```
contract RedemptionBook is Initializable, PermissionedContract, IdSorter, IRedemptionBook {
    ...
    bytes32 constant INITIAL_SUBSCRIPTION_RESTRICTED_PERIOD_ALLOWANCE =
        keccak256("INITIAL_SUBSCRIPTION_RESTRICTED_PERIOD_ALLOWANCE");
    bytes32 constant INITIAL_SUBSCRIPTION_RESTRICTED_PERIOD_FEE =
        keccak256("INITIAL_SUBSCRIPTION_RESTRICTED_PERIOD_FEE");
    bytes32 constant INITIAL_RESTRICTED_PERIOD_ALLOWANCE =
        keccak256("INITIAL_SUBSCRIPTION_RESTRICTED_PERIOD_ALLOWANCE"); // @audit value is clashing with
    ↪ INITIAL_SUBSCRIPTION_RESTRICTED_PERIOD_ALLOWANCE
    ...
}
```

Recommendation(s): Consider changing the value of INITIAL_RESTRICTED_PERIOD_ALLOWANCE to keccak256("INITIAL_RESTRICTED_PERIOD_ALLOWANCE")

Status: Fixed

Update from the client: Fixed in 04dc384

6.18 [Low] Malicious contract manager can continue to pass hasAccess after manager role is changed

File(s): RoleRegistry.sol

Description: In the contract RoleRegistry it is possible to change the unique 32-byte role identifier for the "contract manager" role through the setContractManager(...) function. An expectation of changing the contract manager role identifier is that previous contract managers will no longer have access to the permissioned functions. However, due to a logical or statement in hasAccess(...) it is still possible for an old contract manager with the previous role identifier to continue using permissioned functions. The hasAccess(...) function is shown below:

```
function hasAccess(address _contract, bytes4 _selector, address _requestor, bytes32 _role) public view returns (bool)
{
    Permission memory permission = contractPermissions[_contract].permissions[keccak256(abi.encode(_selector, _role))];
    return hasRole(_role, _requestor)
        && (
            (contractPermissions[_contract].manager == _role)
                || (permission.granted && (permission.timestamp > contractPermissions[_contract].lastReset[_selector]))
        );
}
```

When the contract manager role changes, the line contractPermissions[_contract].manager == _role will always return false, but a malicious manager could have granted access to each function selector individually. This would allow the second logic set in the or statement to be true.

This finding has been assigned a Low severity since it requires no reset, and the Openzeppelin AccessControl role has not been removed for the malicious manager. This would work if the contract admins wanted to change contract managers _only_ by changing the contract manager role identifier, which is very unlikely.

Recommendation(s): After discussion with the client, it seems unlikely that the setContractManager(...) function should ever be used after the contract manager role has been initially set. If this function cannot be called more than once, this issue can be addressed, so consider making changes to setContractManager(...) so that the role identifier can only be set once.

Status: Fixed

Update from the client: Fixed in 9597a11

6.19 [Info] Duplicate elements in `operationModules[]` could cause `deleteModule(...)` to work incorrectly

File(s): RulesEngine.sol

Description: If `IOperationModule(clone).getOperations()` returns duplicated operations, it could cause the function `deleteModule(...)` to work incorrectly. Because the function `deleteModule(...)` has a loop that will break when it finds the first matched element.

```
OPERATIONS[] memory ops = IOperationModule(clone).getOperations();
for (uint256 i = 0; i < ops.length; ++i) {
    for (uint256 j = 0; j < operationModules[ops[i]].length; ++j) {
        // @audit can one module be in ops[i] twice ?
        if (operationModules[ops[i]][j] == clone) {
            //delete address by replacing it with last one in the array and then pop the last one
            operationModules[ops[i]][j] = operationModules[ops[i]][operationModules[ops[i]].length - 1];
            operationModules[ops[i]].pop();
            break;
        }
    }
}
```

Recommendation(s): Consider reviewing the logic for removing modules if duplicated operations are possible within a single module.

Status: Fixed

Update from the client: Fixed in d0cee99

6.20 [Info] Redundant loop check in the function `_editModuleClone(...)`

File(s): DealerRulesEngine.sol

Description: The function `_editModuleClone(...)` is responsible for editing a module. However, it contains a redundant loop check within the code. After finding the position of the module, it breaks the loop. However, since this is a nested loop, the outer loop continues to run and wastes gas.

```
function _editModuleClone(address _moduleClone, bool _status, bytes32 _dealerId) internal {
    OPERATIONS[] memory ops = IOperationModule(_moduleClone).getOperations();
    for (uint256 i = 0; i < ops.length; ++i) {
        for (uint256 j = 0; j < operationModules[ops[i]][_dealerId].length; ++j) {
            //make sure clone belongs to dealer
            if (operationModules[ops[i]][_dealerId][j] == _moduleClone) {
                //enable/disable clone
                // @audit This break will not stop the outer loop
                isDisabledModule[_moduleClone] = !_status;
                break;
            }
        }
    }
}
```

Recommendation(s): Consider returning from the function after finding the position of `_moduleClone` in the function `_editModuleClone(...)`. This will prevent unnecessary execution of the outer loop and optimize gas usage.

Status: Fixed

Update from the client:Fixed in e1de6b8

6.21 [Info] Registering the same module again will not produce the desired outcome

File(s): RulesEngine.sol, DealerRulesEngine.sol

Description: In both rules engines, when a new module is registered, the contract creates a new instance of that module by cloning or deploying it. The function `initialize(...)` is then called on the clone to initialize crucial parameters for the module. Here is the relevant code snippet in Solidity.

```
function _registerModule(address _module, bytes32 _dealerId) internal returns (address) {
    address clone = _module.cloneDeterministic(keccak256(abi.encodePacked(_module, _dealerId)));
    ////////////////////////////////////////////////////////////////
    // @audit add, delete, then add the same module again will not work
    ////////////////////////////////////////////////////////////////
    IOperationModule(clone).initialize(
        investorRegistry, dealerRegistry, fundRegistry, instrumentRegistry, jurisdictionRegistry
    );
    investorRegistry.resetCheckedSinceDealerRulesUpdate(_dealerId);
    OPERATIONS[] memory ops = IOperationModule(clone).getOperations();
    for (uint256 i = 0; i < ops.length; ++i) {
        operationModules[ops[i]][_dealerId].push(clone);
    }
    return clone;
}
```

However, the contract utilizes `CREATE2` to deploy the clone, using the salt as the hash of the original module and dealer ID. Consequently, when the same module is redeployed, it continues to employ the same salt, resulting in an unchanged clone address and subsequent failure.

Recommendation(s): Consider introducing an external parameter to allow for the modification of the salt during the cloning process of the module.

Status: Fixed

Update from the client: Fixed in d0cee99

6.22 [Info] The `total` storage variable in `IdSorter` can be manipulated

File(s): IdSorter.sol

Description: When inserting an ID into the queue using the function `insertId(...)`, it should revert if you insert an ID that already exists. However, if you insert an ID equal to the first ID in the queue, the function will not revert. This will lead to the `total` storage variable being incremented, which is used in `getIdsOrder(...)` and leads to appended zero values on the end of the returned array. This may impact off-chain monitoring or any UI which reads from the function. Because the implementation relies on a linked list with mappings, this does not affect the structure of the queue, so the protocol will continue to work as expected, even with the larger than expected `total`.

This bug is caused by the initial `current` value in the `else` logic. The `current` is set to the slot next to the `start`, i.e., the second value. This means that the comparison to check if the inserted value is equal to `start` never happens. A snipped from the code is shown below:

```
uint256 current = idsQueue[start].next; // @audit `current` is set to the second item in the queue, not first
while (current != 0) {
    if (id < current) {
        idsQueue[idsQueue[current].previous].next = id;
        idsQueue[id].previous = idsQueue[current].previous;
        idsQueue[current].previous = id;
        idsQueue[id].next = current;
        break;
    }
    if (id == current) { // @audit If `current` is equal to `start` this comparison will never be true
        revert IdSorterIDAlreadyInList();
    }
    current = idsQueue[current].next;
}
```

Recommendation(s): Ensure that `start` is checked not to be equal to the new ID to be inserted into the queue.

Status: Fixed

Update from the client: Fixed in b502050

6.23 [Info] Typo in string used to derive storage key

File(s): registries/InstrumentRegistry.sol

Description: The contract `InstrumentRegistry` features eternal storage, using the keccak256 output of a string as the storage key. The key `DEALING_FREQUENCY` contains a keccak256 output from a string containing a typo, as shown below:

```
bytes32 constant DEALING_FREQUENCY = keccak256("DEALING_FRQUENCY");
// "DEALING_FRQUENCY" -> "DEALING_FREQUENCY"
```

This affects the key used when accessing the dealing frequency data.

Recommendation(s): If this code is already deployed, then all further implementation upgrades must feature this same type to ensure that the same storage slot is used, otherwise, empty data may be loaded and affect the protocol. If this code has not been deployed, then there is less risk, and the spelling can simply be corrected.

Status: Fixed

Update from the client: Fixed in 0a237a9

6.24 [Info] `insertId(...)` fails to insert an element equal to the end

File(s): IdSorter.sol

Description: The function `insertId(...)` adds an `id` into the list in ascending order for `idsQueue`. However, if the `id` is equal to `end`, then `total` is incremented, and `id` is not inserted.

```
function insertId(uint256 id) internal {
    if (start == 0) {
        start = id;
        end = id;
    } else if (id > end) {
        idsQueue[end].next = id;
        idsQueue[id].previous = end;
        end = id;
    } else if (id < start) {
        idsQueue[start].previous = id;
        idsQueue[id].next = start;
        start = id;
    } else {
        uint256 current = idsQueue[start].next;
        while (current != 0) {
            if (id < current) {
                idsQueue[idsQueue[current].previous].next = id;
                idsQueue[id].previous = idsQueue[current].previous;
                idsQueue[current].previous = id;
                idsQueue[id].next = current;
                break;
            }
            current = idsQueue[current].next;
        }
    }
    total++;
}
```

Notice that there is **no impact in the current code that inherits this contract since all order IDs are unique**. This is just a highlighting in case the developers **reuse this code in the future for different purposes**.

Recommendation(s): Consider adapting the code to succeed the insertion of `id == end` before `total` is incremented.

Status: Fixed

Update from the client: Fixed in e88db8d

6.25 [Best Practice] Duplicated logic between the `HoldingsModule` and the `BidsAggregationLimitModule`

File(s): `HoldingsModule.sol`, `BidsAggregationLimitModule.sol`

Description: The function `checkSettleBids(...)` in `HoldingsModule` and `BidsAggregationLimitModule` contains duplicated logic when calculating the net holding. This logic can be refactored to improve code cleanliness and avoid repetition.

```
ISecurityToken token = ISecurityToken(instrumentRegistry.getAddress(_instrumentId, INSTRUMENT_TOKEN));
// Adding ((suply - confirmedRedemption) * nav / decimals)
reducedHolding = ( // @audit repeated logic with HoldingsModule, consider refactor
    token.totalSupply()
        - IRedemptionBook(instrumentRegistry.getAddress(_instrumentId, REDEMPTION_BOOK)).getConfirmations()
) * _navPerShare / (10 ** token.decimals());

// ((suply - confirmedRedemption) * nav / decimals) + confirmedSubscriptions
uint256 curHolidng = reducedHolding
    + ISubscriptionBook(instrumentRegistry.getAddress(_instrumentId, SUBSCRIPTION_BOOK)).getConfirmations();

// In HoldingsModules
function _getInvestorNetHoldings(
    bytes32 _investorId,
    ISubscriptionBook _subscriptionBook,
    IRedemptionBook _redemptionBook,
    ISecurityToken _securityToken,
    uint256 _nav,
    uint256 _decimals
) internal view returns (uint256) {
    uint256 investorNetHolding = (
        (_securityToken.getInvestorBalance(_investorId) - _redemptionBook.getInvestorConfirmations(_investorId))
            * _nav
    ) / _decimals;
    investorNetHolding += _subscriptionBook.getInvestorConfirmations(_investorId);
    return investorNetHolding;
}
```

Recommendation(s): Refactor the code by extracting the logic for calculating the reduced holding and getting confirmed subscriptions into separate helper functions. This consolidation will make the code cleaner and remove duplication.

Status: Acknowledged

Update from the client: The logic has some minor changes, and it is not exactly the same anymore.

6.26 [Best Practice] Emitting address instead of index of the wallet

File(s): `BaseUserRegistry.sol`

Description: `removeWallet` function emits the wallet index from the user's wallet address array. But emitting wallet addresses (`walletToDelete`) will be useful and clear.

```
function removeWallet(uint256 _index) external override {
    ...
    address walletToDelete = ownedWallets[_userId][_index];
    ...
    emit WalletRemoved(_index); // @audit emit the address of the removed wallet
    ...
}
```

Recommendation: Consider emitting address of wallet removed (`walletToDelete`) in `removeWallet` function. This provides better information for anyone monitoring or interacting with the smart contract.

Status: Acknowledged

Update from the client: Wallet removals are not going to be possible after fixing investor balance issue

6.27 [Best Practice] Improve efficiency of `insertId(...)` by utilizing a hint for faster position finding

File(s): `IdSorter.sol`

Description: The function `insertId(...)` in the `IdSorter` contract currently loops through a linked list from the start point to locate the position for inserting a new element. However, if the linked list is large, this process can consume a significant amount of gas and potentially exceed the block gas limit.

Recommendation(s): To enhance efficiency, consider implementing a hint mechanism that allows for quicker position finding. The hint must be validated to ensure it is within the queue. The insertion process can be much cheaper by starting the while loop from the hint instead of the beginning.

Status: Fixed

Update from the client: Fixed in 28ff961

6.28 [Best Practice] Including zero check for denominator when performing division

File(s): `BidsAggregationLimitModule.sol`

Description: The `BidsAggregationLimitModule` contains division operations that lack a zero check for the denominator. This can lead to reverting without a clear error message when the denominator is set to 0, resulting in unexpected behavior.

```
uint256 minSubInv = instrumentRegistry.getUint(_instrumentId, MIN_SUB_INV); // @audit no zero check
if (reducedHolding < minHolding) {
    validAmount = minHolding - reducedHolding;
    reducedHolding += _totalInvestment;
    if (reducedHolding > minHolding) {
        validAmount += ((reducedHolding - minHolding) / minSubInv) * minSubInv;
    }
} else {
    reducedHolding = (reducedHolding - minHolding) % minSubInv;
    validAmount = ((reducedHolding + _totalInvestment) / minSubInv) * minSubInv;
    validAmount = (validAmount > reducedHolding) ? validAmount - reducedHolding : 0;
}

function _checkAmount(bytes32 _instrumentId, uint256 _amount) internal view returns (bool) {
    uint256 investmentSize = instrumentRegistry.getUint(_instrumentId, INVESTMENT_SIZE);
    return (_amount % investmentSize) == 0; // @audit zero check for investmentSize
}
```

Recommendation(s): Consider including a zero check for the `investmentSize` denominator before performing the division.

Status: Fixed

Update from the client: Fixed in 0d792a8

6.29 [Best Practice] Missing event emission

File(s): `EternalRegistryStorage.sol`

Description: `_setReservedKey` function in this contract helps to set a reserved key by restricting the key. Several registry contracts use this to set the default reserved keys. It is important to emit an event while this is happening.

```
function _setReservedKey(bytes32 _id, bytes32 _key) internal {
    internalKeys[_id][_key] = true;
    // @audit missing event emission
}
```

Recommendation(s): Consider creating a separate event for adding a new reserved key, for example, `AddedNewReservedKey(bytes32 indexed id, bytes32 indexed key)` and update the `_setReservedKey` function to emit that event.

Status: Fixed

Update from the client: Fixed in d35ec67

6.30 [Best Practice] Prevent initialization of implementation contracts

File(s): `Multiple Contracts`

Description: Currently, it is possible to initialize the implementation contracts, which goes against the intended design. The implementation contracts should not be initialized directly, as they are the underlying logic for other contracts or components.

Recommendation(s): To enforce the intended behavior, consider adding `_disableInitializers(...)` in the constructor of the implementation contracts.

Status: Fixed

Update from the client: Fixed in e88db8d

6.31 [Best Practice] Redundant `investorBalance` getter in `SecurityToken` contract

File(s): `SecurityToken`

Description: The `SecurityToken` contract has a storage variable `investorBalance` which is used to track balances. The variable is `public`, meaning it has its getter function. Another function exists in the contract, named `getInvestorBalance(...)`, which acts as a getter for `investorBalance`. However, since `investorBalance` is already public, this leads to two getters for the same storage variable.

Recommendation(s): Consider either removing the `getInvestorBalance(...)` function or instead the `investorBalance` storage variable to `private` so it will not have its own getter.

Status: Fixed

Update from the client: Fixed in 8e7816d

6.32 [Best Practice] Unnecessary parent before and after token transfer function call

File(s): `SecurityToken.sol`

Description: The `SecurityToken` contract overrides the function `_beforeTokenTransfer(...)` with rules engine checks depending on whether the transfer type is a mint, burn, or normal transfer. In this function, the following line is used:

```
function _beforeTokenTransfer(address _from, address _to, uint256 _amount) internal virtual override {
    // ...

    super._beforeTokenTransfer(_from, _to, _amount);

    // ...
}
```

Using the parent `ERC20Upgradeable._beforeTokenTransfer(...)` function is unnecessary since it is a virtual function without implementation, so no logic will be executed when called. This issue also exists in the `_afterTokenTransfer` function.

Recommendation(s): Consider removing the `super._beforeTokenTransfer(...)` and `super._afterTokenTransfer(...)` lines to remove unnecessary code and improve readability.

Status: Acknowledged

Update from the client: Not applicable anymore after updating OZ's contracts to the latest version and starting using `_update` function.

6.33 [Best Practice] Unnecessary setting of IS_INSTRUMENT and FUND_ID in InstrumentRegistry

File(s): InstrumentRegistry.sol

Description: In the contract InstrumentRegistry, the process of setting up a new instrument is to call addInstrument(...) and then initializeInstrument(...). In the function addInstrument, the IS_INSTRUMENT and FUND_ID storage keys are set for that particular instrument ID. However, these storage keys are set to the same values again in the function initializeInstrument. This leads to two unnecessary storage writes. Code snippets from the two functions are shown below:

```
function addInstrument(bytes32 _senderRole, bytes32 _instrumentId, bytes32 _fundId) external {
    // ...
    _setBool(_instrumentId, IS_INSTRUMENT, true);
    _setBytes(_instrumentId, FUND_ID, _fundId);
    // ...
}

function initializeInstrument(
    ...
) external {
    // ...
    _setBool(_instrumentId, IS_INSTRUMENT, true)
    _setBytes(_instrumentId, FUND_ID, fundId);
    // ...
}
```

Recommendation(s): Consider removing the unnecessary storage writes in initializeInstrument(...).

Status: Fixed

Update from the client: Fixed in e16cb92

6.34 [Best Practice] Unused storage variables

File(s): *.sol

Description: The presence of unused variables is generally considered a practice to be avoided, as they may lead to potential issues such as unnecessary gas consumption and reduced code readability. The following is a list of variables that are unused in the protocol:

- In DealerRegistry the storage variable investorRegistry is set but not used;
- In SecurityToken the storage variable lifecycleContract is not used;
- In InstrumentRegistry the storage variables subscriptionBookImp and redemptionBookImp are not used;

Recommendation(s): Consider removing all unused variables from the codebase to enhance code quality and minimize the risk of errors or inefficiencies.

Status: Fixed

Update from the client: Fixed in 62083e6

6.35 [Best Practice] Verify whether the transfer was successful and the token balance after transfer using order amount

File(s): SubscriptionBook.sol

Description: At the locking tokens phase the ERC20 token is transferred from the sender to the SubscriptionBook contract. In some ERC20 tokens, if the transfer fails, it won't revert instead returns false, or it won't return anything like in USDT. In that case, an order is locked without receiving the tokens.

```
function _afterLockOrderCheck(uint256 _orderId, Order memory _order, address _investorWallet, uint256 _submitter)
↪  internal {
    ...
    paymentToken.transferFrom(sender, address(this), _order.amount);
    // @audit-issue check the return value
    // @audit-issue verify the token balance before and after the transfer using the order amount
    ...
}
```

This applies in case the admin cancels an order. Payment tokens are sent to order beneficiary.

```
function _afterCancelOrderCheck(uint256 _orderId, Order memory _order, uint256 _submitter) internal {
    ...
    if (_order.confirmed) {
        confirmedAmount -= _order.amount;
        confirmedAmountPerInvestor[_order.investorId] -= _order.amount;

        // If locked update the locked amount
        if (!_order.available) {
            amountLocked -= _order.amount;
            paymentToken.transfer(_order.beneficiary, _order.amount);
            // @audit-issue check the return value
            // @audit-issue verify the token balance before and after the transfer using order amount
            removeId(_orderId);
        }
    }
    ...
}
```

Recommendation(s): Consider using SafeERC20 functionalities for transfer and transferFrom. Also, consider comparing the contract's token balances before and after it.

Status: Fixed

Update from the client: Fixed in 520d71c

6.36 [Best Practice] `checkCorrectInvestor(...)` could receive `investorId` instead of `_orderId`

File(s): `SubscriptionBook.sol`

Description: The function `investorConfirmOrder(...)` calls `checkCorrectInvestor(...)` to ensure that the caller is one of the wallets of the order owner. As a further step, in `investorConfirmOrder(...)`, the order is loaded to the memory, as described in the code below.

```
function investorConfirmOrder(uint256 _orderId) public override {
  // @audit The order could be loaded before calling
  //        checkCorrectInvestor(...) and passed investorId instead
  checkCorrectInvestor(_orderId);
  Order memory order = orders[_orderId];
}
```

In addition, `checkCorrectInvestor(...)` reads the `investorId` from the `orders[_orderId]`.

```
function checkCorrectInvestor(uint256 _orderId) internal view {
    if (orders[_orderId].investorId != investorRegistry.getIdFromWallet(msg.sender)) {
        revert IOrderPipelineUnauthorized();
    }
}
```

Recommendation(s): Consider loading the order to the memory before calling `checkCorrectInvestor(...)` and passing the `investorId` as parameter.

Status: Fixed

Update from the client: Fixed in 7473acd

6.37 [Best Practice] `updateInvestorLastOperationTimestamp(...)` parameter can be `investorId`

File(s): `SubscriptionBook.sol` `RedemptionBook.sol`

Description: The function `_afterConfirmOrderCheck(...)` passes the `_orderId` to `updateInvestorLastOperationTimestamp(...)`, which uses the parameter only to get `investidorId` from storage. However, `investidorId` is already loaded in memory. It could be used instead.

```
function updateInvestorLastOperationTimestamp(uint256 _orderId) internal {
  // @audit the function caller has the `investorId`
  investorRegistry.updateInvestorLastOperationTimestamp(
      orders[_orderId].investorId, instrumentId, block.timestamp
  );
}
```

Recommendation(s): Consider passing `_order.investorId` instead since this information is already in memory.

Status: Fixed

Update from the client: Fixed in 7473acd

7 Documentation Evaluation

Software documentation refers to written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

The `Libre` team has provided extensive documentation on their Notion page, including an overview of the protocol and technical insights into its components. A summary of these documents is shown below:

- System Overview
- Subscription Book
- Redemption Book
- Rules Engine
- Registries
- Modules

Each function is complemented by detailed NatSpec comments describing function behavior, inputs and outputs. Moreover, the team conducted a comprehensive code walkthrough and maintained open communication to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

8.1 Tests Output

```
> forge test
Running 5 tests for test/MonotonicQueue.t.sol:MonotonicQueueTest
Test result: ok. 5 passed; 0 failed; finished in 5.05ms
Running 1 test for test/SecurityToken.t.sol:SecurityTokenTest
Test result: ok. 1 passed; 0 failed; finished in 4.70ms
Running 33 tests for test/Allowlist.t.sol:Allowlist
Test result: ok. 33 passed; 0 failed; finished in 23.73ms
Running 8 tests for test/RedemptionCancellationOrders.t.sol:RedemptionCancellationOrdersTest
Test result: ok. 8 passed; 0 failed; finished in 31.23ms
Running 9 tests for test/EternalRegistry.t.sol:BasicEternalRegistry
Test result: ok. 9 passed; 0 failed; finished in 110.30ms
Running 3 tests for test/Gate1Module.t.sol:Gate1ModuleTest
Test result: ok. 3 passed; 0 failed; finished in 126.86ms
Running 3 tests for test/DealerWhitelistedModule.t.sol:DealerWhitelistedModuleTest
Test result: ok. 3 passed; 0 failed; finished in 107.10ms
Running 3 tests for test/Gate3.t.sol:Gate3
Test result: ok. 3 passed; 0 failed; finished in 37.12ms
Running 4 tests for test/RedemptionEndModule.t.sol:RedemptionEndModuleTest
Test result: ok. 4 passed; 0 failed; finished in 188.26ms
Running 2 tests for test/NoticePeriodModule.t.sol:NoticePeriodModuleTest
Test result: ok. 2 passed; 0 failed; finished in 251.21ms
Running 2 tests for test/PhasesManagement.t.sol:PhasesManagementTest
Test result: ok. 2 passed; 0 failed; finished in 15.63ms
Running 2 tests for test/BidCutOffModuleModule.t.sol:BidCutOffModuleTest
Test result: ok. 2 passed; 0 failed; finished in 308.24ms
Running 4 tests for test/HaltModule.t.sol:HaltModuleTest
Test result: ok. 4 passed; 0 failed; finished in 140.57ms
Running 8 tests for test/HoldingsModule.t.sol:HoldingsModuleTest
Test result: ok. 8 passed; 0 failed; finished in 28.59ms
Running 3 tests for test/TotalInvestorsLimitModule.t.sol:TotalInvestorsLimitModuleTest
Test result: ok. 3 passed; 0 failed; finished in 42.45ms
Running 3 tests for test/ForcedRedemptionModule.t.sol:ForcedRedemptionModuleTest
Test result: ok. 3 passed; 0 failed; finished in 634.96ms
Running 2 tests for test/RedemptionTimeCutOff.t.sol:RedemptionTimeCutOffTest
Test result: ok. 2 passed; 0 failed; finished in 6.33ms
Running 2 tests for test/ResetQueueBetweenRounds.t.sol:ResetQueueBetweenRounds
Test result: ok. 2 passed; 0 failed; finished in 25.82ms
Running 3 tests for test/RolesRegistry.t.sol:RoleRegistryTest
Test result: ok. 3 passed; 0 failed; finished in 2.11ms
Running 3 tests for test/RoundAmountLimitModule.t.sol:RoundAmountLimitModuleTest
Test result: ok. 3 passed; 0 failed; finished in 44.24ms
Running 2 tests for test/VolumeLimitModule.t.sol:VolumeLimitModuleTest
Test result: ok. 2 passed; 0 failed; finished in 255.81ms
Running 9 tests for test/SubscriptionCancellationOrders.t.sol:SubscriptionCancellationOrdersTest
Test result: ok. 9 passed; 0 failed; finished in 16.64ms
Running 26 tests for test/AllowlistModuleExample1.t.sol:AllowlistModuleExample1
Test result: ok. 26 passed; 0 failed; finished in 1.06s
Running 2 tests for test/WhitelistedInvestorModule.t.sol:WhitelistedInvestorModuleTest
Test result: ok. 2 passed; 0 failed; finished in 79.12ms
Running 4 tests for test/SubscriptionEndModule.t.sol:SubscriptionEndModuleTest
Test result: ok. 4 passed; 0 failed; finished in 230.62ms
Running 7 tests for test/BidAggregationLimitModule.t.sol:BidsAggregationLimitModuleTest
Test result: ok. 7 passed; 0 failed; finished in 1.22s
Running 15 tests for test/SubscriptionBook.t.sol:SubscriptionBookTests
Test result: ok. 15 passed; 0 failed; finished in 2.07s
Running 13 tests for test/RedemptionBook.t.sol:RedemptionBookTests
Test result: ok. 13 passed; 0 failed; finished in 1.82s
Running 8 tests for test/RedemptionFees.t.sol:RedemptionFeesTest
Test result: ok. 8 passed; 0 failed; finished in 2.54s
Running 5 tests for test/WalletsManagement.t.sol:WalletManagementTest
Test result: ok. 5 passed; 0 failed; finished in 2.34s
```

8.2 Code Coverage

The relevant output is presented below. Please note that the low code coverage for `src/lib` may be due to an issue with Foundry where internal libraries are not accurately tracked (See here). Additionally, the coverage for the code executed within the function `setUp(...)` might not be tracked due to another issue with Foundry (See here).

File	% Lines	% funcs
src/DealerRulesEngine.sol	37.31% (25/67)	43.48% (10/23)
src/EternalRegistryStorage.sol	15.58% (12/77)	42.86% (9/21)
src/IdSorter.sol	63.08% (41/65)	40.00% (2/5)
src/Lifecycle.sol	0.00% (0/3)	0.00% (0/1)
src/OperationModule.sol	0.00% (0/4)	0.00% (0/1)
src/OrderBook.sol	0.00% (0/47)	0.00% (0/7)
src/PermissionedContract.sol	100.00% (3/3)	100.00% (2/2)
src/RedemptionBook.sol	88.44% (306/346)	95.12% (39/41)
src/RulesEngine.sol	52.17% (48/92)	72.22% (26/36)
src/SecurityToken.sol	75.56% (34/45)	75.00% (6/8)
src/SubscriptionBook.sol	90.78% (187/206)	92.31% (36/39)
src/lib/BitMask.sol	0.00% (0/7)	0.00% (0/6)
src/lib/MonotonicQueue.sol	0.00% (0/54)	0.00% (0/8)
src/lib/TimeOperations.sol	0.00% (0/53)	0.00% (0/3)
src/modules/allowlist/AllowlistModuleDealerExample1.sol	38.60% (22/57)	71.43% (5/7)
src/modules/allowlist/AllowlistModuleDealerExampleTesting.sol	100.00% (17/17)	100.00% (8/8)
src/modules/allowlist/AllowlistModuleInstrumentExample1.sol	61.90% (13/21)	60.00% (3/5)
src/modules/allowlist/AllowlistModuleInstrumentExampleTesting.sol	100.00% (17/17)	100.00% (8/8)
src/modules/allowlist/AllowlistModuleLibreExample1.sol	80.00% (40/50)	60.00% (3/5)
src/modules/allowlist/AllowlistModuleLibreExampleTesting.sol	94.12% (16/17)	87.50% (7/8)
src/modules/general/HaltModule.sol	54.17% (26/48)	91.67% (22/24)
src/modules/general/HoldingsModule.sol	87.01% (67/77)	86.67% (13/15)
src/modules/redemptions/ForcedRedemptionModule.sol	70.59% (12/17)	66.67% (2/3)
src/modules/redemptions/Gate1Module.sol	63.64% (7/11)	66.67% (2/3)
src/modules/redemptions/NoticePeriodModule.sol	63.64% (7/11)	66.67% (2/3)
src/modules/redemptions/RedemptionEndModule.sol	37.50% (6/16)	75.00% (6/8)
src/modules/redemptions/VolumeLimitModule.sol	66.67% (8/12)	66.67% (2/3)
src/modules/subscriptions/BidCutOffModule.sol	54.55% (6/11)	80.00% (4/5)
src/modules/subscriptions/BidsAggregationLimitModule.sol	81.82% (27/33)	83.33% (5/6)
src/modules/subscriptions/DealerWhitelistedModule.sol	38.46% (5/13)	80.00% (4/5)
src/modules/subscriptions/RoundAmountLimitModule.sol	14.29% (1/7)	50.00% (2/4)
src/modules/subscriptions/SubscriptionEndModule.sol	44.44% (8/18)	80.00% (8/10)
src/modules/subscriptions/SubscriptionSizeModule.sol	0.00% (0/47)	0.00% (0/10)
src/modules/subscriptions/TotalInvestorsLimitModule.sol	0.00% (0/7)	0.00% (0/4)
src/modules/transfers/WhitelistedInvestorModule.sol	45.45% (5/11)	66.67% (2/3)
src/registries/BaseRegistry.sol	32.14% (18/56)	43.48% (10/23)
src/registries/BaseUserRegistry.sol	96.15% (25/26)	80.00% (4/5)
src/registries/DealerRegistry.sol	78.57% (11/14)	75.00% (3/4)
src/registries/FundRegistry.sol	0.00% (0/7)	0.00% (0/2)
src/registries/InstrumentRegistry.sol	24.24% (24/99)	38.46% (5/13)
src/registries/InvestorRegistry.sol	86.67% (91/105)	73.68% (14/19)
src/registries/JurisdictionRegistry.sol	33.33% (2/6)	40.00% (2/5)
src/registries/RoleRegistry.sol	100.00% (15/15)	100.00% (6/6)
Total	60.16% (1152/1915)	66.35% (282/425)

8.3 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.