certora

Security Assessment Report

CALASTONE

Calastone – Digital Investments Audit

August 2025

Prepared for Calastone

Table of contents

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Initial Commit Hash
Digital Investments	Repo	f6ac48af	172511c69f

Project Overview

This document describes the manual code review findings for **Calastone Digital Investments** project. The work was undertaken from **August 4th** to **August 14th 2025**.

The following contract list is included in our scope:

- evm/contracts/funds/AdminManager.sol
- evm/contracts/funds/ERC2771ContextInitializableUpgradeable.sol
- evm/contracts/funds/Errors/LibErrors.sol
- evm/contracts/funds/Fund.sol
- evm/contracts/funds/FundFactory.sol
- evm/contracts/funds/OrderManager.sol
- evm/contracts/funds/OrderManagerFactory.sol
- evm/contracts/funds/OrderProcessing.sol
- evm/contracts/funds/OrderValidationRules.sol
- evm/contracts/funds/ShareClass.sol
- evm/contracts/funds/Utils/Pausable.sol
- evm/contracts/funds/Utils/Salvage.sol
- evm/contracts/funds/Whitelist.sol
- evm/contracts/funds/interfaces/IFund.sol
- evm/contracts/funds/interfaces/IOrderCommon.sol
- evm/contracts/funds/interfaces/IOrderManager.sol

- evm/contracts/funds/interfaces/IOrderManagerFactory.sol
- evm/contracts/funds/interfaces/IShareClass.sol
- evm/contracts/funds/interfaces/IWhitelist.sol
- evm/contracts/funds/settlement/CalastoneSettlementToken.sol

The team performed a manual audit of all the Solidity contracts in scope**.** During the manual audit, the Certora team discovered bugs in the Solidity contracts code, as listed on the following page.

Protocol Overview

The audit was focused on the **Calastone Tokenised Distribution (CTD) product**, covering the smart contracts in the **DigitalInvestments.Distribution.SmartContracts** repository.

CTD is part of Calastone's digital product suite, operating within an end-to-end fund ecosystem that bridges traditional systems with various blockchain networks (e.g., Ethereum, Avalanche, Polygon).

The product extends Calastone's Transaction Network (CTN) to enable external blockchain connectivity between buyers and sellers of funds, in the same manner that CTN considers connectivity via Swift, API, flat file or other methods. This opens new digital distribution channels for Investment Managers (IMs) and their service providers while preserving existing flows and leveraging their existing integrations with Calastone.

CTD initially focuses on EVM blockchains, with plans to expand.

This phase focuses on allowing IMs (directly or via a service provider) to select share classes to tokenize, choose blockchains, whitelist participants and handle resulting order flow. ERC-20 smart contracts are deployed for Tokenised Fund Share Classes (TFSCs), enabling order submission via smart contracts, mapped to standard order messages and routed to the traditional funds ecosystem via CTN, with responses sent back the same way.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	6	6	6
Low	8	8	6
Informational	26	26	22
Total	40	40	34

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
M-01	Wrong role assigned	Medium	Fixed
M-02	Whitelisting should not be unset	Medium	Fixed
M-03	Incorrect handling of allocatedUnits greater than unitAmount in confirmOrder	Medium	Fixed
M-04	Inconsistent Use of Investment Manager	Medium	Fixed
M-05	CashSell incorrectly handled during order settlement	Medium	Fixed
M-06	Administrator list cannot be updated in FundFactory and Fund contracts	Medium	Fixed
L-01	Insufficient order agent validation	Low	Fixed
L-02	No Return Value Check on transferFrom	Low	Acknowledged
L-03	Current upgradeability pattern could lead to inefficient and inconsistent contract management	Low	Fixed

L-04	Ensure factories are deployed as singletons and store their addresses in a central registry	Low	Acknowledged
L-05	Agent is never checked to be allowed for a particular ShareClass	Low	Fixed
L-06	mintForSettlement reverts if spender has max allowance	Low	Fixed
L-07	Insufficient validation of settlement token addresses	Low	Partially fixed
L-08	Order Manager initialization parameters are swapped	Low	Fixed

Medium Severity Issues

M-01 Wrong role assigned		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: OrderManagerFactory.sol	Status: Fixed	

Description: The setOrderManagerImplementation() function is access controlled by the DEFAULT_ADMIN_ROLE, however the same function requires the UPGRADER_ROLE in all other contracts. Additionally the NatSpec explicitly mentions that UPGRADER_ROLE is expected

```
JavaScript
        * @dev This function updates the address of the order manager implementation
        * and stores it in `_orderManagerImplementation`.
        *
        * Calling Conditions:
        * - Only the "UPGRADER_ROLE" can execute.
        * - The new implementation must support the {IOrderManager} interface.
```

Recommendations: Use the UPGRADER_ROLE role for setOrderManagerImplementation()

Customer's response: Fixed in following commit

"DEFAULT_ADMIN_ROLE is assigned only to addresses used by Calastone to operate the CTD service (this is the same as the UPGRADER_ROLE).

However, as best practice, we accept this recommendation and have removed these setters, which are not required."

Fix Review: Fixed

M-02 Whitelisting should not be unset

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: ShareClass.sol	Status: Fixed	

Description: The project implements a global Whitelist contract which gets called by most of the other contracts to validate the different actors (investors, agents, roles, etc) that interact with them.

Given the highly permissioned nature of the Calastone Tokenized distribution product, it makes sense that whitelisting should not be an optional, but rather a mandatory element of most flows.

Currently all of the contracts allow the whitelist address to be set to address(0) (in initialize & through setter functions). More precisely in the following contracts:

- CalastoneSettlementToken
- Fund
- FundFactory
- ShareClass

Additionally the following whitelisting check is defined on most contracts:

```
JavaScript
function _requireHasAccess(address account) internal view virtual {
            if (address(_whitelist) != address(0)) {
                  if (!_whitelist.hasAccess(account)) {
                        revert LibErrors.AddressNotInWhitelist(account);
                  }
            }
            return;
      }
```

In cases where address(_whitelist) == address(0) (whitelist is not set), _requireHasAccess() must revert , instead of allowing everyone.

Recommendations: Make sure that the whitelist cannot be set to address(0) and revert during validation in case the whitelist address is unset

Note: There is another recommendation that suggests the whitelist address to be moved into a central registry contract, that would be called by all other contracts. In case it gets implemented the suggestions outlined here should be kept in mind

Customer's response: Fixed in following commit

"*Off-chain elements of the CTD service ensure that the global whitelisting contract is always populated as required. However, to strengthen this control, we accept and have implemented the recommendation at the following commit.*"

Fix Review: Fixed

M-03 Incorrect handling of allocatedUnits greater than unitAmount in confirmOrder

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: ShareClass.sol	Status: Fixed	

Description: In `confirmOrder`, when processing `TokenQuantifiedSell` orders, the logic only accounts for the scenario where `allocatedUnits < unitAmount`, in which case the surplus is transferred back to the investor.

JavaScript

```
            if (order.orderType == OrderType.TokenQuantifiedSell && order.orderState ==
OrderState.Accepted) {
                    if (order.submitDetails.unitAmount > confirmData.allocatedUnits) {
                            _lockedFromInvestor[order.submitDetails.investor] -=
(order.submitDetails.unitAmount -
                                    confirmData.allocatedUnits);
                            _transfer(
                                    address(this),
                                    order.submitDetails.investor,
                                    (order.submitDetails.unitAmount -
confirmData.allocatedUnits)
                            );
                    }
            } else if (order.orderType == OrderType.TokenQuantifiedSell && order.orderState
== OrderState.Submitted) {
                    _pullAndLockTokens(order.submitDetails.investor,
confirmData.allocatedUnits);
            }
```

However, the opposite case (`allocatedUnits > unitAmount`) is not handled. This results in **two distinct vulnerabilities** depending on the `orderState`:

1. If the order is already in the `Accepted` state during `confirmOrder` and `allocatedUnits > unitAmount`, the extra tokens are not pulled from the investor and transferred to the share class contract.
    - **Result**: During settlement, the contract attempts to burn allocatedUnits worth of share class tokens. Since it never pulled the excess tokens, it will have insufficient balance, causing the transaction to revert and preventing settlement.
2. If the order is in the `Submitted` state, the function pulls `allocatedUnits` worth of tokens from the investor without checking if this is greater than `unitAmount`.
    - **Result:** This can cause **more share class tokens than the investor intended to sell** to be locked in the contract, violating user expectations and potentially locking funds without consent.

Recommendations: Implement explicit handling for the case where allocatedUnits > unitAmount for both Accepted and Submitted orders.

Customer's response: Fixed in following commit & commit

"*The allocatedUnits amount (i.e. the number of units dealt for an order) is sourced from the IM (or their Transfer Agent, TA) who are the golden source for this information. Where this exceeds the unitAmount (i.e. the number of units requested), this indicates an exceptional flow including logic outside of the blockchain, between the buyer and seller of the tokenised fund (i.e. an off-chain amendment of the order).*

For a redemption, where the allocatedUnits amount exceeds the available token balance for the order sender, then the entirety of that balance will be immobilised (i.e. pulled to the Share Class Contract).

We have implemented this recommendation in order to handle this exceptional scenario in an STP manner. See the following commit."

Fix Review: Fixed

M-04 Inconsistent Use of Investment Manager

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: OrderProcessing.sol	Status: Fixed	

Description: In the current implementation, investmentManager is not stored in the order structure, it is simply passed as a parameter to both confirmOrder and settleOrder in ShareClass contract.

ShareClass.ConfirmOrder()

```
JavaScript
	function confirmOrder(
		string calldata externalId,
		OrderConfirm calldata confirmData,
		address investmentManager
	) external whenNotPaused onlyRole(SHARECLASS_ADMIN) {
		// Code...

		// Mint new tokens ready to swap at settlement
		if (order.orderType == OrderType.TokenQuantifiedBuy || order.orderType ==
OrderType.CashQuantifiedBuy) {
			_mint(investmentManager, confirmData.allocatedUnits);
			uint256 currentAllowance = allowance(investmentManager, _msgSender());
			_approve(investmentManager, _msgSender(), currentAllowance +
confirmData.allocatedUnits);
		}
	}
```

OrderProcessing.settle()

```
JavaScript
function settle(
		IOrderManager orderManager,
		string calldata receiverOrderId,
```

```
            SettlementMechanism memory settlementMechanism,
            address investmentManager,
            IShareClass shareClass,
            address msgSender
        ) internal {
            Order memory order = orderManager.getOrder(receiverOrderId);

            if ((order.orderType == OrderType.TokenQuantifiedBuy || order.orderType ==
OrderType.CashQuantifiedBuy)) {
                // Code ....

                // settle the asset leg
                shareClass.transferWithSpender(
                    investmentManager,
                    order.submitDetails.investor,
                    msgSender,
                    order.confirmDetails.allocatedUnits
                );
            }
        }
```

Because it's not mapped to the order itself, there's no guarantee that the investmentManager address passed during settlement is the same one used during confirmation. This could result in a mismatch where the investmentManager receiving minted shares in confirmOrder is different from the one passed to settle. In such a case, the investmentManager used in settlement might not hold the necessary ShareClass tokens, potentially causing settlement failures or incorrect allocations.

Recommendations: Store the investmentManager address used during acceptOrder/confirmOrder and enforce that the same address is used in settleOrder to prevent inconsistencies.

Customer's response: Fixed in following commit "*Responses to orders (i.e. accepts/rejects, confirmations, and confirmations of fiat settlement) are routed from the traditional funds ecosystem via the off-chain part of the CTD application, which would ensure the same address is used in these cases. Nonetheless, we have implemented this recommendation, via the following commit.*"

Fix Review: Fixed

M-05 CashSell incorrectly handled during order settlement

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: ShareClass.sol	Status: Fixed	

Description: The settleOrder() function of ShareClass executes the following logic:

```
JavaScript
if ((order.orderType == OrderType.TokenQuantifiedSell || order.orderType ==
OrderType.CashQuantifiedSell)) {
                    _lockedFromInvestor[order.submitDetails.investor] -=
order.confirmDetails.allocatedUnits;
             }
```

Problem is that OrderType.CashQuantifiedSell is also handled, which is incorrect, because the logic for Cash related order types has not been implemented in any of the other functions.

Here the code tries to reduce the _lockedFromInvestor mapping, however the mapping is only updated for OrderType.TokenQuantifiedSell in acceptOrder() & confirmOrder() , not for CashQuantifiedSell. As result the _lockedFromInvestor would either be improperly decreased or it will revert

For example if 100 tokens are locked for TokenQuantifiedSell and then 50 allocated CashQuantifiedSell gets settled, it would reduce from the token units, since the cash units have never been locked

Additionally we also have the following handling of CashQuantifiedSell in settle():

```javascript
} else if ((order.orderType == OrderType.TokenQuantifiedSell || order.orderType ==
OrderType.CashQuantifiedSell)) {
    shareClass.burn(address(shareClass), order.confirmDetails.allocatedUnits);
}
```

During settleOrder, for any kind of sellOrders, shareClass tokens are burned. But during acceptOrder and confirmOrder tokens are pulled only for the tokenQuantifiedSell. So, situation could occur where no funds are locked for CashQuantifiedSell but during settlement tokens are burned, leading to dos for other relevant operations

Recommendations: Remove the order.orderType == OrderType.CashQuantifiedSell condition until the Cash logic has been implemented.

Customer's response: Fixed in following commit

"Cash-quantified orders are not part of the current scope of CTD, and their inclusion was forward-looking (and usage was restricted elsewhere in the smart contracts). We accept this recommendation and have removed all relevant logic until this feature is fully implemented."

Fix Review: Fixed

M-06 Administrator list cannot be updated in FundFactory and Fund contracts

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: AdminManager.sol	Status: Fixed	

Description: The FundFactory and Fund contracts contain _administrators storage variables which hold lists of administrator accounts that can perform privileged operations.

This variable is set during initialization to a variable administrators_ by __FundFactory_init and __Fund_init respectively. In addition, during initialization every address in administrators_ is verified to be non-zero and in the whitelist by __AdminManager_init , and thereafter is granted the FUND_ADMIN role.

In AdminManager inherited by FundFactory and Fund contracts, the addAdministrator() and removeAdministrator() functions which grant and revoke the FUND_ADMIN role are gated by the requirement to be DEFAULT_ADMIN_ROLE. However, these functions do not update the _administrators storage variable, contradicting their NatSpecs which say they should update the list of administrators. No other function updates _administrators either.

```JavaScript
function addAdministrator(address newAdmin) external virtual {
            if (newAdmin == address(0)) {
                  revert LibErrors.InvalidAddress();
            }
            _authorizeAdministratorUpgrade();
            _grantRole(FUND_ADMIN, newAdmin);
            emit AdministratorAdded(newAdmin);
      }
```

In FundFactory, the _administrators variable is passed into newly created Fund contracts in createFund() after passing the check that all the addresses are still in the whitelist and non-zero via _verifyAdministrators (which is redundant because the variable cannot be updated).

In Fund, the _administrators variable is passed into newly created ShareClass contracts in createShareclass after passing the check that all the addresses are still in the whitelist and non-zero via _verifyAdministrators (which is redundant because the variable cannot be updated).

Impact: If any address initially passed in as an administrator is removed from the whitelist for some reason, the FundFactory will be unable to create Fund contracts, and Fund will be unable to create ShareClass contracts, leading to a DOS on those flows.

Additionally, if the FUND_ADMIN role is updated there will be a misalignment between the list of addresses which have the FUND_ADMIN role and the list of _administrators which is passed to newly-created Fund and ShareClass contracts.

Recommendations: Override the addAdministrator() and removeAdministrator functions and have them update the list of _administrators. In addition, consider the case where the default admin role can also grant FUND_ADMIN directly without calling addAdministrator() and removeAdministrator.

Customer's response: Fixed in following commit & commit
"*Changing the administrator in this manner is not part of the current scope of CTD, and these functions are not currently used. However we have implemented this recommendation to future-proof the solution, via the following commit.*"

Fix Review: Fixed

Low Severity Issues

L-01 Insufficient order agent validation

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: OrderProcessing.sol	Status: Fixed	

Description: The OrderProcessing.submit() function executes a series of checks that make sure the provided order has valid parameters. One of the checks validates that the caller is either the investor or the agent address like this:

```javascript
if (
submittedOrder.investor != msgSender && (submittedOrder.agent != address(0) &&
submittedOrder.agent != msgSender)
            ) {
                  revert OrderSenderAddressMustBeInvestorOrAgent(
                        senderId,
                        shareClass.getContractAddress(),
                        msgSender,
                        submittedOrder.investor,
                        submittedOrder.agent
                  );
            }
```

And the NatSpec explicitly states that:

```javascript
* - The sender address must be either the investor or the agent.
```

However the current check allows for the following state to be accepted - an order with investor & agent both set to address(0) , would bypass the check - since the agent is checked only if it is **not** a zero address.

This contradicts the documented behavior and should not be allowed. Even though ShareClass which inherits OrderProcessing explicitly checks for address(0) for both fields, it could be problematic in case OrderProcessing gets inherited into other contracts

Recommendations: Always check for the agent address inside, even if empty:

```
JavaScript
if (submittedOrder.investor != msgSender && submittedOrder.agent != msgSender)
        {
              ....
       }
```

Customer's response: Fixed in following commit

"*Agent flows for on-chain order submission will be fully implemented as a roadmap item for CTD. Nonetheless, as best practice, we have implemented this recommendation at the following commit.*"

Fix Review: Fixed

L-02 No Return Value Check on transferFrom

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: OrderProcessing.sol	Status: Acknowledged	

Description: In the settle() function, the result of the transferFrom() call on settlementTokenAddress is not checked.

```JavaScript
            if ((order.orderType == OrderType.TokenQuantifiedBuy || order.orderType ==
      OrderType.CashQuantifiedBuy)) {
                  // settle the cash leg
                  settlementMechanism.settlementTokenAddress.transferFrom(
                        order.submitDetails.investor,
                        investmentManager,
                        order.confirmDetails.settlementAmount
                  );

                  // settle the asset leg
                  shareClass.transferWithSpender(
                        investmentManager,
                        order.submitDetails.investor,
                        msgSender,
                        order.confirmDetails.allocatedUnits
                  );
```

Not all ERC20 tokens revert on failure. If the functions silently fail (e.g., due to allowance or balance issues), the transaction might not revert, leading to partially completed or corrupted settlements. Investors might end up receiving shareClass tokens without transferring settlement tokens.

The current implementation of CalastoneSettlementToken inherits OpenZeppelin's ERC20 contract, which already reverts on failure. However, it is still considered a best practice to explicitly check the returned boolean value for added safety and clarity.

Recommendations: Use safeTransfer method of OpenZeppelin library

Customer's response: *"As noted, the current implementation reverts on failure, due to usage of OpenZeppelin's ERC20 contract, with errors appropriately propagated to ensure atomicity of the transfer, and with appropriate alerting in the off-chain elements of CTD, allowing Calastone Operations team to investigate and resolve.*

Implementation of Safe ERC20 will be considered in future phases of CTD, which may be when fully digital settlement rails are implemented (with the current Settlement Token implementation representing fiat settlement, but futureproof for that planned enhancement)."

Fix Review: Acknowledged

L-03 Current upgradeability pattern could lead to inefficient and inconsistent contract management

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: FundFactory.sol	Status: Fixed	

Description: All of the contracts in the project are upgradeable. They implement the UUPS pattern, by inheriting the relevant contracts from OpenZepplin.

The architecture of the project is such that it involves repeated deployments of the same implementation contract. More precisely this approach is used in the following flows:

- **Fund creation** - FundFactory is used to deploy instances of the Fund contract. The factory stores an updatable _fundImplementation variable and uses it to deploy a new standalone Fund Proxy pointing to the UUPS with that implementation
- **ShareClass creation** - Each Fund stores an updatable _shareClassImplementation and uses it to deploy a new standalone ShareClass Proxy pointing to the UUPS with that implementation
- **OrderManager creation** - OrderManagerFactory is used to deploy instances of the OrderManager contract. The factory stores an updatable _orderManagerImplementation variable and uses it to deploy a new standalone OrderManager Proxy pointing to the UUPS with that implementation
- **CalastoneSettlementToken** - deployed for each currency or payment token again using the UUPS pattern

Given the scale at which Calastone operates it is realistic to assume that the contracts can be deployed in great numbers. Using the UUPS pattern with current architecture is highly inefficient and would create significant management/maintenance challenges as the number of deployments grows.

Currently each deployed contract type (Fund, ShareClass, OrderManager ...) is a separate UUPS contract - meaning it has its individual upgradeability agenda. For example if there are 10 deployed OrderManager contracts, in order to update them the team must go through each proxy one-by-one and invoke upgradeToAndCall .

Another issue is consistency - currently FundFactory, Fund & OrderManagerFactory store the implementation address and update it through set...Implementation function. So in case the implementation gets updated, all newly deployed proxies would use the new version, but the already deployed ones would refer to the old version and would have to be updated manually. This creates a high probability of inconsistency between the Proxy implementations as the their number grows

Recommendations: Implement the Beacon proxy pattern for the following contracts - ShareClass, Fund, OrderManager & CalastoneSettlementToken. Using the Beacon proxy means that the team will have to deploy only a single UpgradeableBeacon that holds the implementation and then the factories would deploy each instance as a BeaconProxy that would point to the singleton UpgradeableBeacon , getting the implementation address from there and delegate calling into it.

When upgrade is required the team would have to call upgradeTo() on the UpgradeableBeacon only once and then all BeaconProxy instances would immediately start using the new implementation. This ensures predictability and consistency.

All other upgradeable contracts that are not meant to be replicated - such as FundFactory, OrderManagerFactory, Whitelist do not require changes and should remain with the current UUPS pattern that they already use.

Customer's response: Fixed in following PR & PR

"We have implemented this recommendation in the following commits."

Fix Review: Fixed

L-04 Ensure factories are deployed as singletons and store their addresses in a central registry

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: FundFactory.sol	Status: Acknowledged	

Description: The FundFactory, OrderManagerFactory & Whitelist contracts are global contracts that are meant to be deployed only once, however the code assumes there could be multiple versions of them, by implementing setters that can update them

- The Fund has the setOrderManagerFactory() and whitelistUpdate() functions
- The FundsFactory has the whitelistUpdate() function
- The ShareClass has whitelistUpdate()
- The CalastoneSettlementToken has whitelistUpdate()

This also makes it more challenging to set the variables across all the different contracts and make sure it is consistent

Recommendations: Consider a new contract that would act as a central registry to hold all global addresses and variables - like FundFactory, OrderManagerFactory, Whitelist. Provide that registry contract to all other contracts to make sure all of them reference the same values. Also updating those variables in the registry once would be enough to update it across all contracts.

In the future this registry can be updated to store any other variables that would be used globally.

Customer's response: *"We acknowledge this recommendation. Currently the "singleton" aspect of these contracts is handled and enforced by the off-chain elements of CTD, as part of contract deployment. We will consider making this change in the future if we deem it necessary."*

Fix Review: Acknowledged

L-05 Agent is never checked to be allowed for a particular ShareClass

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: ShareClass.sol	Status: Fixed	

Description: The submitOrder() and all other order related operations in ShareClass explicitly state in the comments that:

```
JavaScript
* - The agent must be allowed for the share class. {submit}
```

But the agent is never validated against the ShareClass. There is already a check against the global whitelist for agent and investor and in addition the investor is checked against the ShareClass specific _investorAccess whitelist. The documentation creates the assumption that a similar ShareClass access should exist for the agent as well

Recommendations: If agents are expected to be relevant to a particular ShareClass consider implementing a similar check to the one for the investors

Additionally the investor is checked against the local _investorAccess whitelist only in the submit stage and skipped in all other stages

Customer's response: Fixed in following commit & commit

"As noted above, agent flows for on-chain order submission will be implemented as a roadmap item for CTD (and validations here, including with the global whitelist, are already in place).

Nonetheless, to futureproof ahead of that change, we have implemented this recommendation at the following commit."

Fix Review: Fixed

L-06 mintForSettlement reverts if spender has max allowance

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: CalastoneSettlementToken.sol	Status: Fixed	

Description: The mintForSettlement function mints settlement tokens to an account (investor) and then increases the spender's allowance by amount. This allowance update is done as:

```
JavaScript
            _mint(account, amount);
            uint256 currentAllowance = allowance(account, spender);
            _approve(account, spender, currentAllowance + amount);
```

If the investor has **already granted the max allowance** (e.g., `type(uint256).max`) to the spender, which is a common pattern for ERC20 usage, the additional `currentAllowance + amount` will overflow, causing the transaction to revert.

In this protocol, the spender is most likely the ShareClass contract, which will use `transferFrom` during `settleOrder` to collect settlement tokens from the investor.

```
JavaScript
                    settlementMechanism.settlementTokenAddress.transferFrom(
                          order.submitDetails.investor,
                          investmentManager,
                          order.confirmDetails.settlementAmount
                    );
```

Since mintForSettlement already sets the allowance automatically, manual investor approvals are unnecessary. However, an investor unaware of this might approve the ShareClass for type(uint256).max, leading to a subsequent mint failure.

Recommendations: If the currentAllowance is already type(uint256).max, skip the _approve call.

Customer's response: Fixed in following commit

"*As noted, manual approvals are not part of the CTD workflow here. However, we accept and have implemented the recommendation at the following commit.*"

Fix Review: Fixed

L-07 Insufficient validation of settlement token addresses

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: ShareClass.sol	Status: Partially fixed	

Description: Before adding it to the list of ShareClass settlement mechanisms, the `addSettlementMechanism` function only checks that the parameter for the settlement token address is non-zero, emitting `InvalidErc20` if it is. This implies that any non-zero address to be used as a settlement token address is a valid ERC-20. This could allow external and untrusted contracts to be incorporated into an integral part of the system, or cause inconsistencies between settlement token behavior in the same ShareClass.

Recommendations: Settlement tokens should be generated by a factory contract, which is the situation for Fund and ShareClass contracts. `addSettlementMechanism` can then check with the factory that the settlement token was generated by it. This ensures that the settlement token was properly deployed with the correct bytecode, according to the Calastone protocol's specifications.

Customer's response: Partially fixed in following commit - *"We have implemented an ISettlementToken interface to ensure validation of correct implementation of Settlement Tokens and linkages with Share Class Contracts.*

We do not see the factory pattern as necessary for the current implementation of Settlement Tokens, which are solely a means, on a per-currency basis, to reflect fiat settlement, occurring off-chain, on the relevant blockchain."

Fix Review: Partially fixed

L-08 Order Manager initialization parameters are swapped

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: OrderManagerFactory.sol	Status: Fixed	

Description: The createOrderManager function of OrderManagerFactory constructs the initialization data like this:

```
JavaScript
bytes memory initData = abi.encodeWithSelector(IOrderManager.initialize.selector,
orderManagerAdmin, msg.sender);

ERC1967Proxy proxy = new ERC1967Proxy(_orderManagerImplementation, initData);
```

However the parameters of the initialization function are actually defined in reverse:

```
JavaScript
/**
        * @notice Initializes a freshly deployed `OrderManager` proxy.
        * @param defaultAdmin Address that receives `DEFAULT_ADMIN_ROLE`.
        * @param orderManagerAdmin Address that receives `ORDER_MANAGER_ADMIN_ROLE`.
        */
       function initialize(address defaultAdmin, address orderManagerAdmin) external;
```

createOrderManager provides orderManagerAdmin as the first parameter instead of the second. As result orderManagerAdmin is assigned DEFAULT_ADMIN_ROLE & msg.sender ORDER_MANAGER_AMIN_ROLE

Currently createOrderManager is used only by the initialization function of ShareClass, and the interaction with the OrderManager works as intended, despite the discrepancy.

However createOrderManager also works as a standalone function callable by anyone and the documentation and argument parameters create false assumptions about what role the address would actually be assigned.

Recommendations: Rename the input parameter name of createOrderManager() from orderManagerAdmin to defaultAdmin to properly reflect the role being set to that address and also update the comments accordingly

Customer's response: Fixed in following commit

"Use of createOrderManager on a standalone basis is currently not in scope and is restricted via CTD's on-chain Roles-Based Access Control (RBAC) framework. However, we accept the recommendation and have fixed it in the following commit."

Fix Review: Fixed

Informational Issues

I-01. Redundant order state

Description: The OrderState which defines the state transition stages of an order is defined like this:

```
JavaScript
enum OrderState {
              Uninitialized,
              Created, //@audit - not used
              Submitted,
              Accepted,
              Priced,
              Settled,
              Rejected
       }
```

The state Created has not been used anywhere and seems redundant

Recommendation: Remove the Created state option from the enum

Customer's response: Fixed in following commit

Fix Review: Fixed

I-02. Ambiguous trustedForwarder getters

Description: ERC2771ContextInitializableUpgradeable inherits the OpenZeppelin ERC2771ContextUpgradeable contract. It defines a new trustedForwarder_ state variable which substitutes the immutable _trustedForwarder from OZ.

Issue here is that the trustedForwarder() function in the inherited OZ contract still points to _trustedForwarder. While the parent ERC2771ContextInitializableUpgradeable implements a new function getTrustedForwarder() which points to the new variable trustedForwarder_.

This creates ambiguity about the proper forwarder variable version

Recommendation: Consider overriding the trustedForwarder() to point to trustedForwarder_ instead of _trustedForwarder. This would also make the getTrustedForwarder() function redundant

Customer's response: Fixed in following commit

Fix Review: Fixed

I-03. Inconsistent Parameter Naming

Description: Across the OrderProcessing contract, parameters referring to the same logical concept (the order's external identifier) are named inconsistently, e.g., senderId, externalOrderId, externalId, and receiverOrderId. These all represent the order ID in different contexts, but the varying terminology makes the code harder to read and follow.

Recommendation: Use a single, consistent parameter name (e.g., externalOrderId) across all functions where it refers to the same concept to improve clarity and maintainability.

Customer's response: Fixed in following commit & commit

Fix Review: Fixed

I-04. Unused Parameter – investmentManager in acceptOrder

Description: The investmentManager parameter is passed to acceptOrder but is never actually used within the function logic (except for an access check).

```
JavaScript
	function acceptOrder(
		string calldata externalOrderId,
		address investmentManager
	) external whenNotPaused onlyRole(SHARECLASS_ADMIN) {
		Order memory order = _orderManager.getOrder(externalOrderId);
		_requireHasAccess(order.submitDetails.investor);
		_requireHasAccess(order.submitDetails.agent);
		_requireHasAccess(investmentManager);

		if (order.orderType == OrderType.TokenQuantifiedSell) {
			_pullAndLockTokens(order.submitDetails.investor,
order.submitDetails.unitAmount);
		}

		accept(_orderManager, externalOrderId);
	}
```

Recommendation: If the parameter is intended for future use or uniformity across functions, consider documenting its purpose. Otherwise, remove it to simplify the interface.

Customer's response: Fixed in following commit

"*Agreed and fixed in the following commit - see also our response and change on M-04.*"

Fix Review: Fixed

I-05. Interface implementation not checked

Description: The createFund function in the FundFactory contract takes an IOrderManagerFactory orderManagerFactory parameter, however it does not check that orderManagerFactory implements the IOrderManagerFactory interface via ERC165, which is a required condition in the NatSpec.

```
JavaScript
* - `orderManagerFactory` must implement {IOrderManagerFactory} interface.
```

Recommendation: Check that orderManagerFactory implements the IOrderManagerFactory according to ERC165.

Customer's response: Fixed in following commit & commit

Fix Review: Fixed

I-06. Return value of role updating functions not checked

Description: In the AdminManager contract, functions which update FUND_ADMIN role do not check the result of internal functions before emitting superfluous events.

`__AdminManager_init` and `addAdministrator` both call `_grantRole` and immediately emit an `AdministratorAdded` event even though `_grantRole` returns a boolean indicating whether the role was successfully updated.

`removeAdministrator` calls `_revokeRole` and immediately emits an `AdministratorRemoved` event without checking the return value from the function indicating if the role was actually revoked.

This will cause an inconsistency between the events emitted and the result on the program state, and an inconsistency between `AdministratorAdded` and `RoleGranted` , and `AdministratorRemoved` and `RoleRevoked`, events respectively.

This may impact external programs which rely on the `AdministratorAdded` and `AdministratorRemoved` events and their alignment with the `RoleGranted` and `RoleRevoked` events emitted by the OpenZeppelin library functions.

Recommendation: Check the return value of _grantRole and _revokeRole.

Customer's response: Fixed in following commit

"As noted previously - use of these functions is not part of the current scope. However this fix is implemented as a futureproofing point at the following commit."

Fix Review: Fixed

I-07. Insufficient data emitted during share class creation

Description: The createShareclass() function emits an event ShareClassCreated:

```
JavaScript
event ShareClassCreated(string name, address createdAtAddress);
```

The event does not contain the class identifier among the parameters it emits. This is how a ShareClass is identified and stored within a Fund, not the shareClassName (as it is the case for funds).

Recommendation: Add the identifier as parameter in the event, which would also make it easier for offchain indexing

Customer's response: Fixed in following commit

Fix Review: Fixed

I-08. Unused dependencies and variables

Description: The OrderManager contract imports the IWhitelist interface but doesn't use it. Also it defines the _ordersByExternalIdExist state mapping, but doesn't use it in any of its functions

Recommendation: Remove the unused dependency and storage variable

Customer's response: Fixed in following commit

Fix Review: Fixed

I-09. Validate that ShareClass cannot be initialized with an empty baseCurrency

Description: The ShareClass contract does not check in the initializer if the provided baseCurrency_ is not an empty bytes(0) value. Given that there is no way to update the _baseCurrency variable afterwards, it makes sense to ensure that the contract cannot be deployed with an empty value

Recommendation: Check that baseCurrency_ is not bytes3(0)

Customer's response: Fixed in following commit

Fix Review: Fixed

I-10. Settlement mechanisms cannot be disabled

Description: The ShareClass contract enables new settlement mechanisms through addSettlementMechanism() which sets the enabled flag to true.

However there is currently no way to update that flag to false in case the team decides to disable an already enabled settlement mechanism

Recommendation: If disabling a mechanism is an expected functionality by the team, consider introducing a function that can address that.

Customer's response: Fixed in following commit & commit

"*This is not currently supported behaviour - however we have implemented in the following commit and commit, as a futureproofing point ahead of changes to settlement functionality via the product roadmap.*"

Fix Review: Fixed

I-11. Transfer of minted share tokens during settlement is prone to inconsistencies or potential DOS

Description: When an order is confirmed in ShareClass, tokens are minted to the investmentManager and allowance is given to the caller with SHARECLASS_ADMIN role:

```
JavaScript
....
if (order.orderType == OrderType.TokenQuantifiedBuy || order.orderType ==
OrderType.CashQuantifiedBuy) {
                    _mint(investmentManager, confirmData.allocatedUnits);
                    uint256 currentAllowance = allowance(investmentManager, _msgSender());
                    _approve(investmentManager, _msgSender(), currentAllowance +
confirmData.allocatedUnits);
             }
```

In the next stage - settleOrder() - the minted share tokens are sent to the investor through transferWithSpender(), which spends the allowance given during confirmation before transferring them.

The allowance requirement in this case adds additional complexity that is not necessary and can create potential DOS scenarios in the future. Example:

Currently the initializer function takes an array of administrators and assigns the SHARECLASS_ADMIN role to all of them, which creates the assumption that multiple accounts could be granted the SHARECLASS_ADMIN role:

```
JavaScript
for (uint256 i = 0; i < administrators_.length; i++) {
                    if (administrators_[i] == address(0)) {
                           revert LibErrors.InvalidAddress();
                    }
                    _requireHasAccess(administrators_[i]);
                    _grantRole(SHARECLASS_ADMIN, administrators_[i]);
             }
```

We assume the contract has 2 SHARECLASS_ADMIN operators that are used interchangeably.

Operator_1 already has allowance for 100 share tokens, it calls confirmOrder() and receives allowance for 100 more tokens - total of 200.

Operator_2 has allowance for 100 tokens received from order confirmation related to another investor. Operator_2 calls settleOrder() for the current investor (confirmed by Operator_1) and spends his 100 tokens allowance, although they were credited to Operator_1. If Operator_2 tries to settle for the initial investor, it will revert since the allowance was already spent.

Now Operator_1 should switch places and settle it, which could make the process more challenging.

Recommendation: Consider removing the transferWithSpender() flow. Instead just mint the tokens to investmentManager or the ShareClass contract itself and transfer them directly upon settlement. This way it won't matter which SHARECLASS_ADMIN calls the functions

Alternatively you can map the SHARECLASS_ADMIN to an order struct, so that only the relevant SHARECLASS_ADMIN can execute all the actions for a particular order.

Customer's response: Acknowledged

"*We acknowledge this recommendation, but do not see the necessity for any changes.*

Each TFSC (and hence Share Class Contract) will have 1, and only 1, Share Class Admin. This is enforced by the off-chain elements of CTD and how they trigger the deployment of relevant smart contracts, and restrictions on other parties operating these functions are enforced via CTD's whitelisting and on-chain RBAC.

Additional logic would add complexity without meaningful benefit. We can revisit this in the future, should that become necessary."

Fix Review: Acknowledged

I-12. Inconsistent transfer address validation for ShareClass and Settlement Tokens

Description: ShareClass & CalastoneSettlementToken are both an ERC20 token that overrides the transfer(), the transferFrom() & approve() methods to add validation against the global whitelist.

Recommendation: Currently the checks are inconsistent across the 3 functions - while transferFrom() checks both the sender and the receiver of a transfer, the transfer() function does not check the sender (msg.sender). Similarly the approve() function also doesn't validate msg.sender, which would be the sender in a transferFrom() transaction

Customer's response: Fixed in following commit

Fix Review: Fixed

I-13. Insufficient validation in addAdministrator function

Description: When the contract is initialized via __AdminManager_init, each address in initialAdmins is validated against a whitelist through _verifyAdministrators. If an address is not present in the whitelist, initialization reverts. This ensures only approved entities can hold the FUND_ADMIN role.

However, the addAdministrator function bypasses this validation entirely.

```JavaScript
	function __AdminManager_init(address[] memory initialAdmins, IWhitelist whitelist)
internal onlyInitializing {
		__AccessControl_init();
		_verifyAdministrators(initialAdmins, whitelist);
		for (uint256 i = 0; i < initialAdmins.length; i++) {
			_grantRole(FUND_ADMIN, initialAdmins[i]);
			emit AdministratorAdded(initialAdmins[i]);
		}
	}

	function addAdministrator(address newAdmin) external virtual {
		if (newAdmin == address(0)) {
			revert LibErrors.InvalidAddress();
		}
		_authorizeAdministratorUpgrade();
		_grantRole(FUND_ADMIN, newAdmin); // @audit should there be a check wether the
newAdmin has access or not?
		emit AdministratorAdded(newAdmin);
	}
```

Recommendation: Ensure the same whitelist enforcement used in initialization is also applied during administrator addition.

Customer's response: Fixed in following commit

"As noted previously - use of this functions is not part of the current scope. However this fix is implemented as a futureproofing point at the following commit."

Fix Review: Fixed

I-14. Non-compliant ASCII Range Check in _stringToBytes3 Allows Invalid ISO Currency Codes

Description: The _stringToBytes3 function currently validates that each character in the input string is within the ASCII range (0x00–0x7F).

```
JavaScript
function _stringToBytes3(string memory str) internal pure returns (bytes3 result) {
	bytes memory temp = bytes(str);
	if (temp.length != 3) {
		revert LibErrors.InvalidBaseCurrencyLength();
	}
	for (uint256 i = 0; i < 3; i++) {
		if (uint8(temp[i]) > 0x7F) {
			revert LibErrors.NonAsciiBaseCurrencyChar();
		}
		result |= bytes3(temp[i] & 0xFF) >> (i * 8);
	}
}
```

However, ISO currency codes are strictly defined as three uppercase Latin letters (A–Z), which correspond to ASCII values 0x41–0x5A (decimal 65–90).
By allowing any ASCII character up to 0x7F, the function permits invalid currency codes such as lowercase letters (usd), numbers (U5D), or punctuation (U$D)

Recommendation: Restrict the allowed characters to uppercase A–Z by updating the ASCII range check:

```
JavaScript
if (uint8(temp[i]) < 0x41 || uint8(temp[i]) > 0x5A) {
    revert LibErrors.InvalidBaseCurrencyChar();
}
```

Customer's response: Fixed in following commit

Fix Review: Fixed

I-15. OrderManager and OrderManagerFactory are not Pausable

Description: The OrderManager and OrderManagerFactory contracts do not inherit the Pausable contract, which every other concrete contract in Calastone does. This suggests that pausing functionality may be something which is desired to be added to these contracts in the future.

The Pausable contract takes 50 slots of storage, which comes before any initial storage slots that the child uses. If Calastone wants to add pausing functionality to these contracts later on, it will be nearly impossible because the storage slots from the Pausable contract will create a collision with existing storage in the contracts.

Recommendation: Have OrderManager and OrderManagerFactory contracts inherit the Pausable contract, or ensure that pausing functionality will never be desired in these contracts.

Customer's response: Fixed in following commit

Fix Review: Fixed

I-16. Unpaused requirement on some default admin-requiring functions

Description: The following functions are gated by the combination of `whenNotPaused` and `onlyRole(DEFAULT_ADMIN_ROLE)` modifiers:

- Fund.sol: `whitelistUpdate` and `_authorizeAdministratorUpgrade`
- FundFactory.sol: `whitelistUpdate` and `_authorizeAdministratorUpgrade`
- ShareClass.sol: `whitelistUpdate`
- CalastoneSettlementToken.sol: `whitelistUpdate`

In the current configuration, the `whenNotPaused` modifier is not effective as any sort of gate because the DEFAULT_ADMIN_ROLE can grant and revoke all other roles by default, including the PAUSER_ROLE which would allow them to unpause the contract and call these functions.

Recommendation: Clarify what the intended power of the default admin and pauser roles is.

Customer's response: Acknowledged -
"The whenNotPaused modifier intentionally restricts certain functions to execute only when the contract is unpaused, ensuring an immediate fail-fast behavior during a pause.
• *The DefaultAdmin role is explicitly required (onlyRole(DefaultAdmin)) for these functions, ensuring that only an authorized admin can trigger the functionality.*
• *Since DefaultAdmin is already an elevated role governed by the contract's RBAC rules, the current gate provides both the necessary authority and the correct operational context.*

Given these controls, we believe the current implementation meets the intended security and operational requirements without further code changes"

Fix Review: Acknowledged

I-17. Redundant checks on parameters in initialization

Description: During initialization, Fund contracts checks that the whitelist input implements the ERC165 IWhitelist interface; that `defaultAdmin_`, `pauser`, `orderManagerFactory_`, and `shareClassImplementation_` are not zero addresses; and via `__AdminManager_init` that the list of administrators does not contain zero addresses or addresses which are not in the whitelist. The FundFactory contract does the same checks before creating the Fund contract, either in `createFund` or during its own initialization.

ShareClass contracts during initialization check that `defaultAdmin_` and `pauser_` are not zero addresses; and that the list of administrators does not contain zero addresses or addresses which are not in the whitelist. The Fund contract does the same checks before creating the ShareClass contract in the `createShareclass` function.

These redundant checks waste gas since Fund and ShareClass contracts are created atomically by `createFund` and `createShareclass` respectively.

Recommendation: Choose either the creating contract or the created contract to do the check for each specification.

Customer's response: Acknowledged - *"Error handling on a per contract-basis is a better fit for our model. We can review this as a best practices change at a late stage, but this requires no change at this time."*

Fix Review: Acknowledged

I-18. setShareClassImplementation and setOrderManagerFactory do not follow calling conditions

Description: The setShareClassImplementation function in the Fund contract has a `whenPaused` modifier, requiring that the contract is paused, and it does not emit any event. This contradicts the calling conditions which require that the contract not be paused and emit a `ShareClassImplementationUpdated` event when it is successfully updated.

Similarly, `setOrderManagerFactory` also has a `whenPaused` modifier, while its NatSpec specifies that the contract must not be paused.

Recommendation: Align the modifier and the calling conditions, and emit the event after passing the checks.

Customer's response: Fixed in commit

"As per our response to M-01, 'Wrong role assigned', we have deleted these setters and so this finding is now no longer relevant."

Fix Review: Fixed

I-19. Function names do not follow underscoring conventions

Description: In Whitelist.sol, the function `_Whitelist_init` which is called by `initialize` only has one underscore, but internal initialization functions which are called by the initializer should have two (e.g. `__UUPSUpgradeable_init`). The same issue is the case for `_ShareClass_init` in ShareClass.sol and for `_CalastoneSettlementToken_init` in CalastoneSettlementToken.sol.

In OrderValidationRules.sol, two internal functions, `areInstructionAmountsValid` and `validateStateChange` , lack underscore prefixes which they should have as internal functions in a contract.

In OrderProcessing.sol, the internal functions `submit`, `accept`, `confirm`, `settle`, and `reject` do not have underscore prefixes which internal functions should have according to convention.

Recommendation: Consider addressing the above recommendations

Customer's response: Fixed in following commit

Fix Review: Fixed

I-20. Incomplete parent initializer calls risk upgrade incompatibility

Description: Multiple contracts inherit from OpenZeppelin upgradeable base contracts but fail to call their corresponding initializer functions, breaking the expected initialization chain.

1. __ERC165_init() not called
    - Missing in: Fund, OrderManager, ShareClass, and Whitelist.
2. __AccessControl_init() not called
    - Missing in: OrderManager and OrderManagerFactory.

Currently, both init functions are empty, so there's no immediate impact of the issue, but during the future upgrades, If OpenZeppelin updates them to include initialization logic, it could cause problems. Additionally, the initialization sequence will be incomplete, violating the pattern's best practice of calling all relevant parent initializers.

Recommendation: Call all relevant parent initializers in each contract

Customer's response: Fixed in following commit

Fix Review: Fixed

I-21. Access Checks and Zero Address Validation Missing in _CalastoneSettlementToken_init

Description: The _CalastoneSettlementToken_init initializer has two issues:

1. Missing Zero Address Check for trustedForwarder:
    - The NatSpec specifies that trustedForwarder must not be the zero address, but the function does not enforce this requirement.
2. Ineffective Access Checks for defaultAdmin and pauser:
    - The function calls _requireHasAccess(defaultAdmin) and _requireHasAccess(pauser) before the _whitelist variable is set. Since _whitelist is still address(0) at this point, _requireHasAccess returns early without performing any real whitelist validation. Furthermore, _whitelist is explicitly set to the zero address later in the function, making these checks permanently ineffective in this call.

This means the intended access control guarantees during initialization are not enforced.

Recommendation:

1. Add a zero address validation for trustedForwarder.
2. Pass the whitelist address as a parameter to _CalastoneSettlementToken_init and set _whitelist before performing any _requireHasAccess checks.

Customer's response: Fixed in following commit

Fix Review: Fixed

I-22. Typos in NatSpecs

Description: There are many NatSpecs with calling conditions which have seeming typos which contradict the behavior of the described functions:

1. In the FundFactory contract, `__FundFactory_init` NatSpec requires that `defaultAdmin_`, `pauser`, and `fundImplementation_` "must not be a non-zero address," but the function instead checks that these ARE non-zero addresses.
2. The setFundImplementation function's NatSpec says `Contract must not be paused` but the function has a whenPaused modifier, which conditions execution on the contract BEING paused.
3. In the Fund contract, `__Fund_init` NatSpec requires that `defaultAdmin_`, `pauser`, `orderManagerFactory_`, and `shareClassImplementation_` "must not be a non-zero address." But all of these parameters are checked for BEING non-zero addresses.
4. In the ShareClass contract, `__ShareClass_init` NatSpec says `defaultAdmin_`, `orderManagerAdmin`, `pauser`, and `orderManagerFactory_` "must not be a non-zero address." But all these parameters are checked for BEING non-zero addresses.
5. The ShareClass contract describes the `_settlementMechanism` mapping as `* @notice Maps each share class to its settlement mechanism.` but it maps a settlement currency or token, not a share class, to its settlement mechanism.

Recommendation: Align the function behavior and NatSpec calling conditions.

Customer's response: Fixed in following commit , commit, commit

Fix Review: Fixed

I-23. OrderState to string conversion misses a state

Description: In the `OrderValidationRules` contract, the function `convertStateToString` converts OrderState enum names to matching strings, except OrderState.Uninitialized values are converted to the string “Unknown” which covers values outside the range of the enum.

Recommendation: Convert OrderState.Uninitialized states to the string “Uninitialized” or some other string to differentiate them from truly unknown enum values.

Customer’s response: Fixed in following commit

Fix Review: Fixed

I-24. Missing Validation for CashQuantified Orders

Description: The function areInstructionAmountsValid is designed to validate order amounts based on the OrderType, specifically distinguishing between TokenQuantified and CashQuantified orders. However, it only handles TokenQuantified cases (i.e., TokenQuantifiedBuy and TokenQuantifiedSell) and does not implement any validation for CashQuantified order types (CashQuantifiedBuy, CashQuantifiedSell) as mentioned in the natspec For the CashQuantified order types it checks if the unit amount is 0 and the currency amount is valid.

```
JavaScript
        // It reverts if currency is not supported by the ShareClass contract. For the
CashQuantified order types it checks
        // if the unit amount is 0 and the currency amount is valid. For the TokenQuantified
order types it checks if the
        // currency amount is 0 and the unit amount is valid.
```

Recommendation: Extend the areInstructionAmountsValid() function to handle CashQuantified order types.

Customer's response: Fixed in following commit

"As per the response to M-05, 'CashSell incorrectly handled during order settlement', cash-quantified order logic has been removed, and this finding is therefore no longer relevant."

Fix Review: Fixed

I-25. Share class feature statuses emit superfluous events

Description: In a ShareClass contract, the `setSubscription`, `setRedemption`, and `setCashOrder` functions state in their NatSpecs that they emit events when they change the enabled/disabled status of their respective features. However, they emit events even when the status does not change.

Recommendation: Only emit events when the statuses change, or modify the NatSpecs to align with the function behaviors.

Customer's response: Fixed in following commit

Fix Review: Fixed

I-26. Pausable modifier added on implementation() view only function

Description: Adding modifiers on non state changing functions is not a common and recommended design approach, since it can distort the user experience (also it might be problematic for blockchain indexers) and sometimes lead to unexpected behaviour if not planned correctly.

Currently the team has implemented a pausable modifier to the `implementation()` function on the `Beacon`, which is a getter all proxies call to get the implementation address. In case it gets paused all proxies would be disabled and also no one would be able to check the address of the implementation

Recommendation: Consider removing the modifier from the view function

Customer's response: Acknowledged - *"The ability to pause all contracts en masse, rather than on a per-contract basis, is an important control.*

We therefore feel that the benefits of this change outweigh the potential costs (which could be remedied by unpausing), and we will therefore keep the implementation as is."

Fix Review: Acknowledged

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.