

Security Assessment Report



Calastone CTD

February 2026

Prepared for Calastone

certora

CALSTONE
An SS&C Company

Table of Contents

Project Summary **3**
- Project Scope 3
- Project Overview 3
- Protocol Overview 4

Assessment Methodology **6**

Threat and Security Overview **7**
- Findings Summary 9
- Severity Matrix 9

Detailed Findings **10**
- Medium Severity Issues 12
  - M-01: Improper Investor/Agent role separation enabling unauthorized order submit and transfers 12
  - M-02: Sell Orders May Become Non-Rejectable Due to Global Locked Token Burning 14
- Low Severity Issues 19
  - L-01: Missing access verification for pauser during Settlement token initialization 19
  - L-02: Insufficient Access Control in transferFrom in SettlementToken allowing whitelist bypass 20
  - L-03: _confirmAllocation() reverts if the spender has a max allowance 22
  - L-04: Transfers Can Bypass investorIsAllowed and _transferEnabled Checks via Partially Allowed Endpoints 24
- Informational Severity Issues 26
  - I-01: Unbounded array growth and gas griefing via duplicate administrator entries 26
  - I-02: Unsafe uint256 to int256 typecast in settleOrder may overflow event amounts 28
  - I-03: Access Control for submitConfirmedOrder Differs from Documented Role 29
  - I-04: Missing Disabled Token Check for settlementTokenAddress in _settle 30
  - I-05: Missing whitelist check for investmentManager in submitConfirmedOrder 31
  - I-06: Typo in OrderManager Role Name 32
  - I-07: Unbounded Iteration May Exceed Block Gas Limit 33

Disclaimer **34**

About Certora **34**

certora

CALASTONE
An SS&C Company

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Calastone CTD	https://github.com/calastone/DigitalInvestments.Distribution.SmartContracts/tree/main	141905d	Solidity

Project Overview

This document describes the security review of **Calastone CTD**. The work was undertaken from **February 4th, 2026** to **February 16th, 2026**.

The following contract list is included in our scope:

```
evm/contracts/Proxy/Proxy.sol
evm/contracts/Proxy/BeaconProxyWrapper.sol
evm/contracts/Proxy/Beacon.sol
evm/contracts/funds/AdminManager.sol
evm/contracts/funds/ERC2771ContextInitializableUpgradeable.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/Whitelist.sol
evm/contracts/funds/Errors/LibErrors.sol
evm/contracts/funds/settlement/CalastoneSettlementToken.sol
evm/contracts/funds/Utils/Pausable.sol
evm/contracts/funds/Utils/Salvage.sol
```

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

Protocol Overview

The Calastone Tokenised Distribution (CTD) project is a strategic expansion of the Calastone Transaction Network designed to integrate traditional investment fund systems with public blockchain environments like Ethereum, Avalanche, and Polygon.

Calastone's Tokenised Distribution is a full smart-contract suite and tooling set for managing tokenized investment funds on EVM networks. It models a complete fund lifecycle: creating funds and share classes, whitelisting investors, submitting/validating/settling buy & sell orders, and handling settlement tokens.

At the core, a central Fund Factory deploys individual Fund contracts, which in turn manage multiple ShareClass instances that function as ERC-20 compliant tokens representing an investor's holdings. Each share class is supported by a dedicated Order Manager that enforces a strict state machine - advancing orders from submission and acceptance through pricing and final settlement - to ensure institutional-grade compliance and security. Security is further reinforced through a robust on-chain whitelisting mechanism, role-based access controls, and emergency pause functions, all while supporting meta-transactions to provide a user-friendly, gasless experience for authorized participants.

This review focused primarily on three major changes introduced in this update:

1. Support for off-chain orders via the introduction of `submitConfirmedOrder`, which submits and confirms an order within a single transaction – enabling digital distributors/investors to initiate orders in tokenised share classes via a traditional connection into the Calastone Transaction Network (i.e. via Swift, API, flat file), as an additional option to integrating with CTD smart contracts and initiating orders on-chain.

2. A configurable `transferEnabled` flag that allows transfers between whitelist distributors when enabled – adding additional control for Investment Managers or their service providers (i.e. Transfer Agents), on top of existing protection of transfers via whitelisting.

3. Introduction of batch unpausing functionality – enhancing pause functionality reviewed in the prior review.

In addition to the above, the review also covered several smaller changes and refactors included in this update.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

Threat and Security Overview

The Calastone Tokenised Distribution (CTD) system is a hierarchical, role-based fund management platform that bridges traditional finance with EVM blockchains. Its security model is built on a 'defense-in-depth' strategy, utilizing strict on-chain whitelisting, upgradeable proxy patterns, and circuit-breaker mechanisms to manage the lifecycle of Tokenised Fund Share Classes (TFSCs).

System Model

The system consists of a central **FundFactory** that manages **Fund** proxies, which in turn oversee individual **ShareClass** tokens. Security relevant elements include:

• **Trust Boundaries**: The primary boundary exists between the **Whitelist** and the other contracts, no asset movement or order submission is permitted for accounts not validated by this contract. A secondary boundary exists between the TrustedForwarder(meta-transactions) and the internal `_msgSender()` logic.

• **Security Assumptions**: It is assumed that the `DEFAULT_ADMIN_ROLE` and `UPGRADER_ROLE` are held by secure multi-sig wallets or governance contracts, as they control the implementation logic and forwarder addresses.

• **Invariants**: Key invariants include:

1. The total number of locked tokens for an investor must never exceed their `balanceOf`.

2. Orders must follow a linear state progression: `Submitted` → `Accepted` → `Priced` → `Settled` (note – the Accepted state can be skipped, and orders progress from Submitted to Priced, where supported by the relevant Investment Manager and Transfer Agent, and per their off-chain flow of messages into the Calastone Transaction Network).

3. Only the `ShareClass` contract itself can burn its own tokens to prevent unauthorized supply reduction.

Threat Models and Scenarios Considered

As part of the security assessment, we consider a range of potential attack vectors that commonly apply to smart contract systems of this design. The following categories represent

threat models evaluated during the review and do not necessarily correspond to specific vulnerabilities in the current implementation. Any confirmed issues identified during the review are documented in the Findings section below.

- **Gas Exhaustion via Loop Injection**: Exploiting the `pauseAllShareClasses` or `pauseAllFunds` functions by creating an excessive number of identifiers, causing the administrative loops to exceed the block gas limit and preventing emergency stops.
- **Dust Donation**: Forcing ETH into contracts to potentially disrupt accounting logic, although the Salvage module provides a mechanism to recover such funds.
- **Broken Pausing**: Critical state-changing functions, such as initialize or specific administrative setters, lacking the `whenNotPaused` modifier, allowing configuration changes during an active emergency.
- **Meta-Transaction Spoofing**: Exploiting a compromised or poorly validated `TrustedForwarder` to forge the `_msgSender()`, allowing an attacker to submit orders on behalf of a whitelisted investor.
- **Inconsistent Access Control**: Bypassing the Whitelist if internal functions (like `_transfer`) are called by a logic path that forgets the `_requireHasAccess` check.
- **Re-initialization**: Exploiting initialization functions that may lack proper protection or idempotency, allowing an attacker to re-run setup logic and overwrite the `defaultAdmin` or whitelist addresses.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	2	2	2
Low	4	4	2
Informational	7	7	4
Total	13	13	8

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	Improper Investor/Agent role separation enabling unauthorized order submit and transfers	Medium	Fixed
M-02	Sell Orders May Become Non-Rejectable Due to Global Locked Token Burning	Medium	Fixed
L-01	Missing access verification for pauser during Settlement token initialization	Low	Acknowledged
L-02	Insufficient Access Control in transferFrom in SettlementToken allowing whitelist bypass	Low	Acknowledged
L-03	`_confirmAllocation` reverts if the spender has a max allowance	Low	Fixed
L-04	Transfers Can Bypass investorIsAllowed and _transferEnabled Checks via Partially Allowed Endpoints	Low	Fixed
I-01	Unbounded array growth and gas griefing via duplicate administrator entries	Informational	Acknowledged
I-02	Unsafe uint256 to int256 typecast in settleOrder may overflow event amounts	Informational	Fixed

I-03	Access Control for submitConfirmedOrder Differs from Documented Role	Informational	Fixed
I-04	Missing Disabled Token Check for settlementTokenAddress in _settle	Informational	Acknowledged
I-05	Missing whitelist check for investmentManager in submitConfirmedOrder	Informational	Fixed
I-06	Typo in OrderManager Role Name	Informational	Fixed
I-07	Unbounded Iteration May Exceed Block Gas Limit	Informational	Acknowledged

Medium Severity Issues

M-01: Improper Investor/Agent role separation enabling unauthorized order submit and transfers

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: evm/contracts/funds/ShareClass.sol	Status: Fixed	

Description:

In ShareClass, the same mapping is currently used for both investors and agents, which fails to enforce proper role separation. As a result:

- If an address is registered as an agent, calling `investorIsAllowed` may still return `true`.
- Likewise, if an address is registered as an investor, calling `agentIsAllowed` may also return `true`.

This overlap in role validation leads to several issues:

1. An investor can act as an `agent` and create orders on behalf of other investors, which contradicts the intended model where an investor should only be able to create orders for themselves.

   This behavior can enable spam or misuse, even if such orders are eventually rejected by the `SHARECLASS_ADMIN`.

   Additionally, an investor could DOS an agent's `submitOrder` transaction by front-running it and creating a fake order with the same `externalOrderId`, causing the transaction to revert with `ExternalOrderIdNotUnique`.

2. Incorrect transfer behavior may also occur. A share class that is intended to be transferable only to investors could instead be transferred to an address acting as an agent. This weakens the transfer restrictions.

Recommendations:

Introduce separate mappings for investors and agents in ShareClass.

Customer Response: Fixed in commit fd9bbd7

"Agent flows for on-chain order submission will be fully implemented as a roadmap item for CTD and their current inclusion in smart contract code is forward looking and to futureproof CTD smart contracts.

For now, we have ensured that inclusion does not introduce adverse consequences, by adding additional validations to ensure that the Investor Address and Agent Address on an on-chain order submission are the same, preventing agency-style flows for these orders. Additionally, in the current product version, we will never be calling the "allowAgentAccess" function, and this is controlled by the off-chain CTD application. Therefore, all whitelisted addresses will be for investors, not agents, and hence permitted to hold tokens.

We will introduce fuller role separation as part of the full implementation of agency-style flows to CTD."

Fix Review:

Fixed

M-02: Sell Orders May Become Non-Rejectable Due to Global Locked Token Burning

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: evm/contracts/funds/ShareClass.sol	Status: Fixed	

Description:

During settlement of a **token-quantified sell order**, the contract determines the burn amount based on the order's confirmed allocation, but caps it using the **total locked balance of the investor**, rather than the portion locked specifically for that order:

```
uint256 burnAmount = order.confirmDetails.allocatedUnits;
uint256 availableToBurn =
shareClass.getLockedFromInvestor(order.submitDetails.investor);

// for sell orders when confirmed allocated units is more than the investor's
balance,
if (burnAmount > availableToBurn) {
    burnAmount = availableToBurn;
}

// burn only when there's something to burn
if (burnAmount > 0) {
    shareClass.burn(address(shareClass), burnAmount);
}
```

As a result, settlement can burn **all tokens locked by the investor**, including tokens that were locked to back **other active sell orders**, not just the tokens corresponding to the order being settled.

This becomes problematic when multiple sell orders from the same investor are active concurrently.

A concrete failing scenario is demonstrated by the provided test case:

1. A first sell order is submitted and **confirmed with** `allocatedUnits` **exceeding the investor's available balance**
2. Before settlement, the same investor submits and progresses a second sell order, which locks tokens
3. The first sell order is then settled, burning all locked tokens for the investor
4. When attempting to reject the second sell order, the protocol tries to unlock tokens that have already been burned
5. This causes an **underflow in the locked balance accounting**, and the rejection reverts

As a result, the second order becomes stuck in an unrejectable state.

POC

```
          it("should fail to reject second order after first order with
over-allocated units is settled", async function () {
                const { shareClassExposed, otherAdmin, investor1, investor2,
whitelist } = await loadFixture(deployAndSubmitOrderFixture);

                // investor1 has 10000 tokens from fixture
                const initialBalance = await
shareClassExposed.balanceOf(investor1.address);
                console.log("Initial balance:",
ethers.formatEther(initialBalance));

                // First sell order (order2 from fixture is already submitted for
100 tokens)
                const firstOrderId = "firstSellOrder";
                const secondOrderId = "secondSellOrder";

                // Submit first sell order
                await shareClassExposed.connect(investor1).submitOrder(
                     firstOrderId,
                     1, // TokenQuantifiedSell
                     {
                          unitAmount: ethers.parseEther("100"),
```

```
                        currencyAmount: 0,
                        instructionCurrencyOrToken: "USD",
                        investor: investor1.address,
                        agent: investor2.address,
                        orderDateTime: 0,
                        requestedSettlementCurrencyOrToken: "TEST",
                    },
                );

                // Confirm first order with MORE than available (12000 > 10000
available)
                const overAllocatedUnits = ethers.parseEther("12000");
                await shareClassExposed.connect(otherAdmin).confirmOrder(
                    firstOrderId,
                    {
                        unitPrice: 1000000,
                        allocatedUnits: overAllocatedUnits,
                        grossAmount: 10,
                        netAmount: 10,
                        feeAmount: 10,
                        settlementAmount: 10,
                        expectedSettlementDate: 0,
                        tradeDateTime: "2023-10-01T00:00:00Z",
                    },
                    otherAdmin.address
                );

                const balanceAfterFirstConfirm = await
shareClassExposed.balanceOf(investor1.address);
                console.log("Balance after first confirm:",
ethers.formatEther(balanceAfterFirstConfirm));

                // Submit second sell order before first is settled
                await
shareClassExposed.connect(otherAdmin).$__mint(investor1.address,
ethers.parseEther("1000"));
                await shareClassExposed.connect(investor1).submitOrder(
                    secondOrderId,
                    1, // TokenQuantifiedSell
```

```
                    {
                        unitAmount: ethers.parseEther("100"),
                        currencyAmount: 0,
                        instructionCurrencyOrToken: "USD",
                        investor: investor1.address,
                        agent: investor2.address,
                        orderDateTime: 0,
                        requestedSettlementCurrencyOrToken: "TEST",
                    },
                );

                // Accept second order (this locks tokens)
                await
shareClassExposed.connect(otherAdmin).acceptOrder(secondOrderId,
otherAdmin.address);

                const balanceAfterSecondAccept = await
shareClassExposed.balanceOf(investor1.address);
                console.log("Balance after second accept:",
ethers.formatEther(balanceAfterSecondAccept));

                // Settle first order
                await
shareClassExposed.connect(otherAdmin).settleOrder(firstOrderId,
otherAdmin.address);

                const balanceAfterFirstSettle = await
shareClassExposed.balanceOf(investor1.address);
                console.log("Balance after first settle:",
ethers.formatEther(balanceAfterFirstSettle));

                // Now try to reject second order - this should fail due to
underflow in _lockedFromInvestor
                // because first order consumed all the locked tokens
                await expect(

shareClassExposed.connect(otherAdmin).rejectOrder(secondOrderId, "Rejection
reason")
```

```
            ).to.be.reverted;
        });
```

Recommendations:

Consider tracking **locked balances on a per-order basis**, rather than using the investor's total locked amount.

Customer Response: Fixed in commit [fd9bbd7]

"We acknowledge the potential for this edge-case scenario.

Rather than tracking locked tokens by order, as per the recommendation, we have updated the logic for rejected redemptions, so that the amount of tokens unlocked is capped at the total available for the investor (ensuring the action will not fail). This mirrors the approach introduced for price confirmation and settlement of redemptions, which caps immobilisation and burning of tokens at the available investor balance.

In both cases, this is because the source of truth for unit ownership, and for responses to orders (acceptance/rejection, price confirmation, and confirmation of fiat settlement) is the Transfer Agent and their unit register, rather than the on-chain token balance."

Fix Review:

Fixed

Low Severity Issues

L-01: Missing access verification for pauser during Settlement token initialization

Severity: Low	Impact: Low	Likelihood: Low
Files: evm/contracts/funds/settlement/CalastoneSettlementToken.sol	Status: Acknowledged	

Description:

The __CalastoneSettlementToken_init function in CalastoneSettlementToken.sol includes documentation and a logical requirement that both the defaultAdmin and pauser addresses must have access to the token (i.e., be present in the whitelist). However, the implementation only executes the _requireHasAccess(defaultAdmin) check.

The pauser address is granted the PAUSER_ROLE immediately after. Because the pauser is not verified against the _whitelist during initialization, an address that is not authorized to interact with the system could be assigned a critical operational role.

Recommendations:

Modify the __CalastoneSettlementToken_init function to include an access check for the pauser address, ensuring consistency with the defaultAdmin check.

Customer Response: Acknowledged

"Settlement tokens are a means, on a per-currency basis, to reflect fiat settlement, occurring off-chain, on the relevant blockchain. They do not bear any value and their inclusion is to futureproof CTD for the addition of future digital settlement rails. Accordingly, whilst we acknowledge this recommendation, we do not see a change as necessary now, and will revisit it as part of a fuller implementation."

Fix Review:

Acknowledged

L-02: Insufficient Access Control in transferFrom in SettlementToken allowing whitelist bypass

Severity: Low	Impact: Medium	Likelihood: Low
Files: evm/contracts/funds/settlement/CalastoneSettlementToken.sol	Status: Acknowledged	

Description:

The transferFrom function does not enforce _requireHasAccess(_msgSender()) when msgSender is a contract that supports the IShareClass interface.

```
          if (
               _msgSender().code.length == 0 ||
!IERC165(address(_msgSender())).supportsInterface(type(IShareClass).interfaceId)
          ) {
               _requireHasAccess(_msgSender());
          }
```

This appears to be intended so that share classes can call transferFrom without being whitelisted, but the current implementation allows any contract that mimics the interface to bypass the whitelist. If a contract is whitelisted and later removed from the whitelist, it can still call transferFrom function, contrary to the intended whitelist only action model. The only requirement is that both the from and to addresses are whitelisted..

Recommendations:

Restrict this behavior by explicitly maintaining and checking a registry of trusted share class contracts allowed to call transferFrom without whitelist checks, rather than relying on interface support alone

Customer Response: Acknowledged

"Settlement tokens are a means, on a per-currency basis, to reflect fiat settlement, occurring off-chain, on the relevant blockchain. They do not bear any value and their inclusion is to

futureproof CTD for the addition of future digital settlement rails. Accordingly, whilst we acknowledge this recommendation, we do not see a change as necessary now, and will revisit it as part of a fuller implementation."

Fix Review:

Acknowledged

L-03: _confirmAllocation() reverts if the spender has a max allowance

Severity: Low	Impact: Medium	Likelihood: Low
Files: evm/contracts/funds/ShareClass.sol	Status: Fixed	

Description:

In the _confirmAllocation function, when the orderType is TokenQuantifiedBuy, the contract mints the confirmed amount to the Investment Manager(IM) and subsequently increases the Calastone admin's allowance by that same amount.

```
            if (orderType == OrderType.TokenQuantifiedBuy) {
                  _mint(investmentManager, confirmed);
                  uint256 currentAllowance = allowance(investmentManager,
_msgSender());
                  _approve(investmentManager, _msgSender(), currentAllowance +
confirmed);
            }
```

In standard ERC20 workflows, users frequently grant type(uint256).max as an allowance to trusted entities to minimize gas costs. If the Investment Manager has already granted this maximum allowance to the Calastone admin, the addition of any non-zero amount will trigger an arithmetic overflow this results in transaction to revert.However, if the Investment Manager is unaware of this behavior, they may grant the Calastone admin an allowance of type(uint256).max, potentially resulting in failures in confirmOrder and submitConfirmedOrder calls.

Recommendations:

If confirmed exceeds `type(uint256).max - currentAllowance`, cap the allowance increase at `type(uint256).max - currentAllowance` to prevent overflow.

Customer Response: Fixed in commit fd9bbd7

“In the current product iteration, both of these roles and addresses will be managed via CTD, which will ensure the described scenarios are avoided.

Nonetheless, we acknowledge these findings, relevant for a future state where clients may manage their smart contracts directly, where appropriate. We have made changes to cater for when the allowance is set to "max", and additional scenarios will be handled as part of future smart contract changes.”

Fix Review:

Fixed

L-04: Transfers Can Bypass investorIsAllowed and _transferEnabled Checks via Partially Allowed Endpoints

Severity: Low	Impact: Medium	Likelihood: Low
Files: evm/contracts/funds/ShareClass.sol	Status: Fixed	

Description:

The transfer logic allows an address that is **whitelisted** but **not marked as allowed via** investorIsAllowed(...) to successfully call transfer() and send tokens to an allowed recipient. This situation can arise if an address was previously included in the allow list and later removed, while remaining whitelisted.

This occurs because the _update() hook only enforces the _transferEnabled gate when **both the sender and the recipient are marked as allowed**. As a result a sender that is whitelisted but not investorIsAllowed can still initiate a transfer as long as the recipient is an allowed address

In this scenario, the transfer proceeds without enforcing _transferEnabled, effectively bypassing the intended restriction.

```
     function _update(address from, address to, uint256 amount) internal virtual
override {
          if (investorIsAllowed(from) && investorIsAllowed(to)) {
               if (!_transferEnabled) revert TransfersNotEnabled();
          }

          super._update(from, to, amount);
     }
```

Additionally, the NatSpec documentation for transfer() states that the sender must be allowed to send tokens:

```
      * @dev Calling Conditions:

      * - The `sender` is allowed to send tokens.
```

```
      * - The `to` is allowed to receive tokens.

      */
     function transfer(address to, uint256 amount) public override whenNotPaused
returns (bool) {
```

However, in the current implementation, transfer() does not validate investorIsAllowed(_msgSender()). The only sender-side check performed is that the sender is whitelisted.

Recommendations:

If the intention is to restrict transfers, consider enforcing _transferEnabled directly within transfer() and transferFrom()

Customer Response:

Fixed in commit fd9bbd7

Fix Review:

Fixed

Informational Severity Issues

I-01: Unbounded array growth and gas griefing via duplicate administrator entries

Files:

evm/contracts/funds/Fund.sol

Description:

The _addAdministrator function in Fund.sol overrides the base AdminManager logic to synchronize the _administrators storage array. However, it fails to check if the newAdmin already possesses the FUND_ADMIN role before pushing to the array. While the underlying role assignment is idempotent, the array is not, repeated calls with the same address will cause the _administrators array to grow indefinitely.

Because there is no validation to prevent adding an existing administrator, a malicious or accidental caller with the DEFAULT_ADMIN_ROLE can repeatedly add the same address to the system. This results in several critical issues:

1. The `_administrators` array can grow indefinitely with redundant data.
2. Functions that iterate over the administrators, such as `_removeFromAdministratorsArray`, will consume increasingly more gas, potentially leading to Out-of-Gas reverts and a Denial of Service.
3. The `getAdministrators` view function will return a list containing duplicate addresses, providing an inaccurate representation.

Recommendations:

Modify the _addAdministrator function in Fund.sol to verify that the newAdmin does not already possess the FUND_ADMIN role before executing the addition and array push logic

Customer Response: Acknowledged

"The "administrator" flow caters for a future use case where Transfer Agents, or other service providers to Investment Managers, may administer tokenised share classes directly on-chain, rather than via the CTD application, where appropriate. This is not part of the current scope, and

the inclusion is deliberate, forward looking and futureproof. Accordingly, we acknowledge this recommendation and will consider making this change as a part of a full implementation."

Fix Review:

Acknowledged

I-02: Unsafe uint256 to int256 typecast in settleOrder may overflow event amounts

Files:

evm/contracts/funds/OrderManager.sol

Description:

The settleOrder function performs unchecked typecasts from uint256 to int256 for allocatedUnits and settlementAmount. If either value exceeds type(int256).max, the cast will overflow and produce an incorrect signed value. This impacts the OrderSettled event emission, potentially misleading off-chain consumers and creating accounting or audit discrepancies, even if on-chain state updates remain correct.

Recommendations:

Use OpenZeppelin's SafeCast to perform validated conversions and revert on overflow

Customer Response:

Fixed in commit fd9bbd7

Fix Review:

Fixed

I-03: Access Control for submitConfirmedOrder Differs from Documented Role

Description:

The NatSpec documentation for submitConfirmedOrder states that the function should only be callable by an account with the SHARECLASS_ADMIN role:

```
 * - The caller must have the "SHARECLASS_ADMIN" role
```

However, the function is actually protected using the CALASTONE_ADMIN_ROLE:

```
function submitConfirmedOrder(
    ...
) external whenNotPaused onlyRole(CALASTONE_ADMIN_ROLE) {
```

As a result, the implemented access control does not match the documented calling conditions.

Recommendations:

If the current role restriction is intentional, consider updating the NatSpec documentation to reflect CALASTONE_ADMIN_ROLE.Alternatively, if the function is intended to be restricted to SHARECLASS_ADMIN, consider aligning the role check with the documentation.

Customer Response: Fixed in commit fd9bbd7

"The current role restriction is intentional and a different role than SHARECLASS_ADMIN is purposefully used for responses to off-chain orders. The NatSpec documentation has been updated to reflect the CALASTONE_ADMIN_ROLE."

Fix Review:

Fixed

I-04: Missing Disabled Token Check for settlementTokenAddress in _settle

Files:

evm/contracts/funds/ShareClass.sol

Description:

In the _settle function, the contract transfers settlementAmount from investor without verifying that this token address is not disabled.This allows settlements to proceed using tokens that should no longer be active or allowed, weakening intended token controls and potentially enabling settlement with unsupported or deprecated assets.

Recommendations:

Before transferring settlementAmount, add an explicit check to ensure settlementTokenAddress is not disabled

Customer Response: Acknowledged

"Settlement tokens are a means, on a per-currency basis, to reflect fiat settlement, occurring off-chain, on the relevant blockchain. They do not bear any value and their inclusion is to futureproof CTD for the addition of future digital settlement rails. Accordingly, whilst we acknowledge this recommendation, we do not see a change as necessary now, and will revisit it as part of a fuller implementation.

In particular here, we do not have a current use case for disabling these settlement tokens (but may for that future digital settlement feature)."

Fix Review:

Acknowledged

I-05: Missing whitelist check for investmentManager in submitConfirmedOrder

Files:

evm/contracts/funds/ShareClass.sol

Description:

submitConfirmedOrder does not validate whether the investmentManager is whitelisted, whereas acceptOrder performs this whitelist validation.

This creates an inconsistency in the checks and allows orders to progress further in the workflow without the expected whitelist enforcement, even if the validation may occur later during settlement.

Recommendations:

Add an explicit whitelist validation for investmentManager in submitConfirmedOrder, matching the checks performed in acceptOrder.

```
        _requireHasAccess(investmentManager);
```

Customer Response:

Fixed in commit fd9bbd7

Fix Review:

Fixed

I-06: Typo in OrderManager Role Name

Files:

evm/contracts/funds/OrderManager.sol

Description:

The OrderManager contract uses a misspelled role constant ORDER_MANAGER_AMIN_ROLE instead of ORDER_MANAGER_ADMIN_ROLE, which can confuse maintainers and causes code inconsistency.

Recommendations:

Rename the role identifier to ORDER_MANAGER_ADMIN_ROLE and update all references accordingly across contracts.

Customer Response:

Fixed in commit fd9bbd7

Fix Review:

Fixed

I-07: Unbounded Iteration May Exceed Block Gas Limit

Files: evm/contracts/funds/Fund.sol

Description:

The functions pauseAllFunds, unpauseAllFunds, pauseAllShareClasses, and unpauseAllShareClasses iterate over all existing fund or share class identifiers and perform per-entry checks and external calls.

Since Ethereum enforces a block gas limit of ~30 million gas, transactions that iterate over a sufficiently large number of entries may exceed the available gas and revert. As these functions do not implement batching or pagination, their gas usage scales linearly with the total number of funds or share classes.

In practice, reaching a number of entries large enough to cause a denial of service is unlikely, as it would require a very high number of funds or share classes. Additionally, even if such a situation were to arise, an account with the pauser role could still pause or unpause individual contracts directly.

Nevertheless, it is highlighted primarily to make the team aware of the potential limitation should the protocol scale to a large number of funds or share classes in the future.

Recommendations:

If the team anticipates managing a large number of funds or share classes, consider introducing **batched or paginated pause/unpause functions** that operate over a specified index range. This would improve operational convenience and ensure that global pause operations remain usable regardless of system growth.

Customer Response: Acknowledged

"We will revisit this recommendation as the CTD product matures, and volumes increase (current projected usage indicates this will not be an issue for some time and remediation will be planned as needed)."

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.