Tradable onchain v2 diff

Security Review

Cantina Managed review by:
Cergyk, Security Researcher
Akshay Srivastav, Security Researcher

November 8, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Tradable enables leading asset managers to safely and compliantly tokenize their strategies and access a rapidly growing and underserved global market for institutional grade yield products.

From Oct 29th to Oct 31st the Cantina team conducted a review of onchain-v2-diff on commit hash dab60f64. The team identified a total of **10** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 5
- Low Risk: 1
- Gas Optimizations: 1
- Informational: 3

3 Findings

3.1 Medium Risk

3.1.1 `redemptionQueueTotal` not decreased during `_decreaseRedemptionAmount` could lead to overestimation of `availableForInvestment`

Severity: Medium Risk

Context: InvestmentManager.sol#L621

Description: The variable `redemptionQueueTotal` tracks the total amount contained in the redemption queue. It is used to determine the amount still "investable" in the deal, through the `availableForInvestment` function:

- InvestmentManager.sol#L181-L194:

```
/// @notice Returns the max amount that can be invested in the deal.
function availableForInvestment() public view returns (uint256) {
    IDeal dealToken = deal();
    uint256 totalSupply = dealToken.totalSupply();
    // If the deal is open-ended, it is effectively unlimited.
    if (dealToken.isOpenEnded()) return type(uint256).max - totalSupply;

    // Calculate the available emission.
    uint256 totalSize = dealToken.totalSize();
    uint256 availableEmission = totalSize > totalSupply ? totalSize - totalSupply : 0;

    // If the deal is not open-ended, the investment cap is the sum of the available emission,
    // the deal token balance of the manager contract and the total amount in the redemption queue.
    // @audit: redemptionQueueTotal() is added to investable amount
    return availableEmission + dealToken.balanceOf(address(this)) + redemptionQueueTotal(); // <<
}
```

Unfortunately, the variable `redemptionQueueTotal` is not updated when `_decreaseRedemptionAmount` applies a partial decrease to a queued redemption:

- InvestmentManager.sol#L601-L626:

```
/// @notice Decreases the redemption request amount if the investor's balance is not sufficient.
function _decreaseRedemptionAmount(address investor, uint256 newTokenBalance) internal {
    RMStorage storage $ = _rmStorage();
    // Check whether investor has an active redemption request.
    (bool exist, uint256 id) = $.investorRedemptionRequests.tryGet(investor);
    if (!exist) return;

    // Check if the balance dropped to zero. Delete the redemption request if so.
    //!audit-ok just delete redemption request
    if (newTokenBalance == 0) {
        _deleteRedemptionRequest(id);
        return;
    }

    // Retrieve the redemption request.
    RedemptionRequest storage rr = $.redemptionRequests[id];

    // The adjustable amount depends on the redemption request phase.
    uint256 rrAmount = rr.queued ? rr.availableAmount : rr.amount;
    if (newTokenBalance < rrAmount) {
        emit RedemptionAmountUpdated(id, rrAmount, newTokenBalance);
        if (rr.queued) {
            //@audit: redemptionQueueTotal should be updated
            rr.availableAmount = newTokenBalance; // <<
        } else {
            rr.amount = newTokenBalance;
        }
    }
}
```

Impact: Not decreasing `redemptionQueueTotal` correctly will lead to an overestimation of `availableForInvestment`, and may mislead the keepers into calling `reviewOffer` with an `acceptedAmount` too high, thus failing the transactions unexpectedly.

Recommendation: Consider reducing `redemptionQueueTotal` as follows:

- InvestmentManager.sol#L601-L626:

```
/// @notice Decreases the redemption request amount if the investor's balance is not sufficient.
function _decreaseRedemptionAmount(address investor, uint256 newTokenBalance) internal {
    // ...

    // Retrieve the redemption request.
    RedemptionRequest storage rr = $.redemptionRequests[id];

    // The adjustable amount depends on the redemption request phase.
    uint256 rrAmount = rr.queued ? rr.availableAmount : rr.amount;
    if (newTokenBalance < rrAmount) {
        emit RedemptionAmountUpdated(id, rrAmount, newTokenBalance);
        if (rr.queued) {
            rr.availableAmount = newTokenBalance;
+           $.redemptionQueueTotal -= (rrAmount - newTokenBalance);
        } else {
            rr.amount = newTokenBalance;
        }
    }
}
```

Tradable: Fixed in commit 05b143c0.

Cantina Managed: Fixed.

3.1.2 `burnDealTokens` can brick queued redemptions processing

Severity: Medium Risk

Context: DealManager.sol#L239-L242

Description: The `DEAL_ADMIN` role is able to directly burn tokens from any holder by calling on `DealManager::burnDealTokens`.

- DealManager.sol#L239-L242:

```
/// @inheritdoc IDealManager
function burnDealTokens(IDeal.Burn[] calldata targets) external restricted returns (uint256
↪  totalBurned) {
    totalBurned = deal().burn(targets);
    emit TokensBurned(totalBurned);
}
```

However any of the holders may have a queued redemption. And as a result of the burn, if the holder has not enough balance for his redemption to be processed, the whole redemption queue will be stuck:

- InvestmentManager.sol#L697-L707:

```
// Calculate the amount to redeem, ensuring it doesn't exceed the available liquidity.
// NOTE: We can compare tokens and payment currency amounts directly as they have the same decimals.
uint256 redeemedAmount = FixedPointMathLib.min(rr.availableAmount, availableLiquidity);
// Consume the available liquidity.
availableLiquidity -= redeemedAmount;

// Transfer redeemed deal tokens. The redeemed tokens must always go to the manager contract first.
// @audit: transfer fails here if rr.investor has insufficient balance
dealToken.managedTransfer(rr.investor, address(this), redeemedAmount); // <<
if (redeemer != address(this)) {
    dealToken.managedTransfer(address(this), redeemer, redeemedAmount);
}
```

This is correctly handled in `_initiatePrincipalPayout`, by calling `_decreaseRedemptionAmount`:

- DealManager.sol#L351-L354:

```
/// @notice Principal payout callback function.
function _afterPrincipalPayout(IPayoutManager.PrincipalPayout memory payout) internal {
    _decreaseRedemptionAmount(payout.account, payout.newTokenBalance);
}
```

Impact: The redemption processing queue ends up permanently blocked.

Note that `DEAL_ADMIN` can unblock processing by minting tokens to `rr.investor`, but it should not be correct in practice (rr.investor may have already received redemption off-chain for example).

Recommendation: `_decreaseRedemptionAmount` should be called for every burn target:

- DealManager.sol#L239-L242:

```
    /// @inheritdoc IDealManager
    function burnDealTokens(IDeal.Burn[] calldata targets) external restricted returns (uint256
↪   totalBurned) {
+       for (uint256 i = 0; i<targets.length; ++i) {
+           uint targetBalance = deal().balanceOf(targets[i].from);
+           _decreaseRedemptionAmount(targets[i].from, targetBalance-targets[i].amount);
+       }
+
        totalBurned = deal().burn(targets);
        emit TokensBurned(totalBurned);
    }
```

Tradable: Fixed in commit d9fe89a2.

Cantina Managed: Fixed.

3.1.3 Partial redemptions will fail if max holders is reached and deal manager's dealToken balance is zero

Severity: Medium Risk

Context: InvestmentManager.sol#L522, InvestmentManager.sol#L704-L707

Description: During redemptions, a `managedTransfer` is made to `DealManager` before transferring deal tokens to "final" redeemer. In some scenarios, partial redemptions would revert due to the `DEAL_MAX_HOLDERS` requirement, despite the fact `DealManager` is only a temporary holder.

- InvestmentManager.sol#L704-L707:

```
dealToken.managedTransfer(rr.investor, address(this), redeemedAmount);
if (redeemer != address(this)) {
    dealToken.managedTransfer(address(this), redeemer, redeemedAmount);
}
```

Indeed when applying `_afterBalanceUpdate`, no distinction is made for the `DealManager` address, so it is added as a regular holder, and can trigger the `MaxHoldersExceeded` error.

- Deal.sol#L690-L697:

```
if (balanceBefore == 0 && balanceAfter > 0) {
    // The balance is changed from zero, add the new holder, respecting the maximum holders limit.
    uint256 maxHoldersLimit = $.maxHolders;
    if ($.currentHolders.length() >= maxHoldersLimit) revert MaxHoldersExceeded(maxHoldersLimit);

    assert($.currentHolders.add(account));
    emit HolderStatusUpdated(account, true);
} else if (balanceBefore > 0 && balanceAfter == 0) {
```

Proof of Concept:

- Preconditions: `DEAL_MAX_HOLDERS = 2`

1. Alice and Bob are two holders which initially sign into the deal by providing 1M$ each. `DealManager` holds zero tokens.
   - Scenario 1: `redemptionBudget = 0`
2. Alice wants out of the deal by 500k$, and thus submits a redemption request.
3. Bob submits an offer for 500k$ expecting to "buy" the deal tokens from Alice.
4. When the deal admin calls `reviewOffer` for the full amount, it reverts, because Alice's deal tokens are first transferred to `DealManager`, thus attempting to add `DealManager` address as an additional holder.

    - Scenario 2: `redemptionBudget = 500k`
5. Alice wants out of the deal by 500k$, and thus submits a redemption request.
6. When deal admin calls `reviewRedemptionRequest`, the call fails because the deal tokens must be transferred to `DealManager`.

Recommendation: `DealManager` is a special address with regards to holders of the deal tokens, because it needs to hold tokens to handle some operations such as redeems. Consider adding the special case to `_afterBalanceUpdate`:

- Deal.sol#L680-L702:

```
    function _afterBalanceUpdate(address account, uint256 balanceBefore) private {
        // Ignore the zero address.
        if (account == address(0)) return;

        // If the balance did not change, there is no need to update the holder status.
        uint256 balanceAfter = balanceOf(account);
        if (balanceBefore == balanceAfter) return;

        DealStorage storage $ = _storage();
        // Update the holder status.
+       if (account != $.manager) {
            if (balanceBefore == 0 && balanceAfter > 0) {
                // The balance is changed from zero, add the new holder, respecting the maximum holders
↪   limit.
                uint256 maxHoldersLimit = $.maxHolders;
                if ($.currentHolders.length() >= maxHoldersLimit) revert
↪   MaxHoldersExceeded(maxHoldersLimit);

                assert($.currentHolders.add(account));
                emit HolderStatusUpdated(account, true);
            } else if (balanceBefore > 0 && balanceAfter == 0) {
                // The balance is changed to zero, remove the holder.
                assert($.currentHolders.remove(account));
                emit HolderStatusUpdated(account, false);
            }
+       }
    }
```

Tradable: Fixed in PR 62.

Cantina Managed: Fixed.

3.1.4 Incorrect `PayoutManager.payoutDustThreshold` implementation leads to more tokens getting burned than expected

Severity: Medium Risk

Context: PayoutManager.sol#L144-L149

Description: The `PayoutManager.payoutDustThreshold` function intends to return `0.01` token equivalent of deal token, but currently it returns `1` token equivalent.

```
function payoutDustThreshold() public view returns (uint256) {
    return 0.01e4 * (10 ** uint256(paymentCurrency().decimals() - 2));
}
```

The function currently returns 100X more value than the expected dust amount.

Impact: Anything below `1` deal token balance of investors is now considered as dust and gets burned in `_principalPayoutDistribution` function call.

Recommendation: Consider changing the implementation to this:

```
  function payoutDustThreshold() public view returns (uint256) {
-     return 0.01e4 * (10 ** uint256(paymentCurrency().decimals() - 2));
+     return 10 ** uint256(paymentCurrency().decimals() - 2);
  }
```

Also the calculations in developer comments of the function also need to be corrected.

Tradable: Fixed in commit e9bacbf2.

Cantina Managed: Fixed.

3.1.5 Redemption requests can be reviewed more than once

Severity: Medium Risk

Context: InvestmentManager.sol#L484-L490

Description: The `InvestmentManager._reviewRedemptionRequest` function currently lacks protection against reviewing a redemption request more than once. Due to this a single redemption request can be reviewed multiple times by the `DEAL_ADMIN`, leading to unintended scenarios and accounting issues.

Consider a redemption request of 100 tokens which got accepted, 50 tokens were redeemed from redemption budget and 50 tokens were queued. For this request, the `DEAL_ADMIN` can perform any of these operations:

- Call the `_reviewRedemptionRequest` again with `acceptedAmount` as `0` and delete the queued request of 50 tokens.
- If liquidity is present then call the `_reviewRedemptionRequest` with `acceptedAmount` as `100`, in this case 100 tokens will be redeemed instantly.
- If liquidity is not present then call the `_reviewRedemptionRequest` with `acceptedAmount` as `100`, in this case 100 tokens will be queued in redemption queue.

Recommendation: Consider adding a check in `_reviewRedemptionRequest` function which reverts if the request is already queued.

Tradable: Fixed in commit 36e81568.

Cantina Managed: Fixed.

3.2 Low Risk

3.2.1 Missing `nonReentrant` modifier on `_reviewRedemptionRequest` function

Severity: Low Risk

Context: InvestmentManager.sol#L529

Description: The `InvestmentManager._reviewRedemptionRequest` function does not follow the checks-effect-interaction pattern and performs external call before updating its internal state.

`paymentCurrency().safeTransfer(...)` is performED at InvestmentManager.sol#L529 before calling `_deleteRedemptionRequest(...)` at InvestmentManager.sol#L550.

Note that to perform re-entrancy `paymentCurrency` should be a token with hooks (ERC777) and `investor` should be `DEAL_ADMIN` and a smart contract. The likelihood of this happening is very low.

Recommendation: Consider adding `nonReentrant` modifier to `_reviewRedemptionRequest` function.

Tradable: Fixed in commit a2a82966.

Cantina Managed: Fixed.

3.3 Gas Optimization

3.3.1 Pushing payout to DealManager can skip unecessary transfer call

Severity: Gas Optimization

Context: PayoutManager.sol#L284

Description: When calling `DealManager::pushPayout`, the receiver can be `address(this)` because `DealManager` can hold tokens temporarily and thus receive yield payouts. In that case an unecessary call to `transfer` can be skipped.

Recommendation:

```
- paymentCurrency.safeTransfer(account, amount);
+ if (account != address(this))
+     paymentCurrency.safeTransfer(account, amount);
```

Tradable: Acknowledged.

Cantina Managed: Acknowledged.

3.4 Informational

3.4.1 Incorrect `InvestmentManager.redemptionRequests` function implementation

Severity: Informational

Context: InvestmentManager.sol#L243-L262

Description: The `InvestmentManager.redemptionRequests` function intends to return all active redemption requests. While iterating over `investorRedemptionRequests` it tries to skip deleted requests, but still stores the request data at index `i` in `requests` variable which is incorrect.

```
for (uint256 i = 0; i < requests.length; i++) {
    (, uint256 id) = $.investorRedemptionRequests.at(i);
    RedemptionRequest storage rr = $.redemptionRequests[id];
    // Skip deleted redemption requests.
    if (rr.id == 0) continue;

    requests[i] = rr;
    requestCount++;
}
```

As per the `for` loop's context the request data should be stored at index `requestCount` instead of `i`. Moreover, looking at `_deleteRedemptionRequest()` function implementation there should be no need to skip deleted requests in `redemptionRequests` function as `investorRedemptionRequests` should not be storing any deleted requests.

Recommendation: Consider simplifying the “ implementation to this:

```
function redemptionRequests() public view returns (RedemptionRequest[] memory requests) {
    RMStorage storage $ = _rmStorage();
    requests = new RedemptionRequest[]($.investorRedemptionRequests.length());
    for (uint256 i = 0; i < requests.length; i++) {
        (, uint256 id) = $.investorRedemptionRequests.at(i);
        requests[i] = $.redemptionRequests[id];
    }
}
```

Tradable: Fixed in commit d338ad48.

Cantina Managed: Fixed.

3.4.2 `InvestmentManager._setMinInvestment` uses unoptimal constant for `minInvestment_` validation

Severity: Informational

Context: InvestmentManager.sol#L268-L269

Description: The `InvestmentManager._setMinInvestment` function compares `minInvestment_` amount with `BasisPointsLib.DENOMINATOR` constant (1e4) which have no logical co-relation with each other.

Recommendation: Ideally minimum investment amount should be compared with a dynamic payment currency decimals scaled value. Or, atleast should be compared with a static custom constant value.

Tradable: Fixed in commit e6b48220.

Cantina Managed: Fixed.

3.4.3 No order is enforced on offer review and may favor some participants

Severity: Informational

Context: InvestmentManager.sol#L336-L341

Description: Deposits are escrowed until the deal admin reviews them, only after that deal tokens are sent to the investor. The investor may be subject to the `originationFee`, if there are not enough tokens in the `DealManager` contract or queued redemptions to service the deposit (e.g if tokens have to be minted).

As a result, when multiple deposits are waiting to be reviewed, and tokens need to be minted for part of the deposits, the deal admin can decide on which investors pay the fee by changing the order of review of the offers.

- Charlie's redeem request for 500k$ is queued.
- Alice and Bob submit deposits for 500k$ each.

Deal admin now can choose which of Alice or Bob gets the tokens by processing Charlie's redeem, and avoid paying the `originationFee`.

> This may result in an unfair outcome if a policy of first-come-first-serve is implied, but not enforced

Recommendation: Consider implementing a tracking of `lastReviewedOfferId` and being able to review the offers only sequentially

- InvestmentManager.sol#L332-L341:

```
/// @notice Review an investment offer.
/// @dev Handles the offer review and emits the `OfferReviewed` event.
/// @param offerId The offer ID.
/// @param acceptedAmount The amount of the offer that is accepted.
function _reviewOffer(uint256 offerId, uint256 acceptedAmount) internal {
    IMStorage storage $ = _imStorage();
    if (offerId == 0 || !$.offerIds.contains(offerId)) revert OfferNotFound();

+   if (offerId != lastReviewedOfferId + 1) revert OutOfOrderOfferReview();
    // Retrieve and cache offer details to reduce storage reads.
    InvestmentOffer memory offer = $.investmentOffers[offerId];
```

Tradable: Acknowledged.

Cantina Managed: Acknowledged.