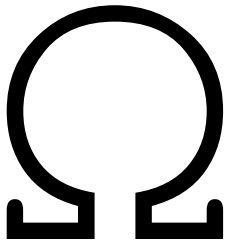


Team Omega

https://teamomega.eth.limo

Karpatkey KPK Shares Audit

Final Report

October 13, 2025

Summary

Karpatkey has asked Team Omega to audit a contract that implements the logic for a shared fund.
The current document is a preliminary audit report. The Karkatkey team will read this and fix any issues they choose to fix, and Omega will subsequently review the fixes and write a final report.

Team Omega
Team Omega (https://teamomega.eth.limo/) specializes in Smart Contract security audits on the Ethereum ecosystem.

Karpatkey
Karpatkey (kpk, https://kpk.io/) helps organisations secure, manage, and grow their onchain assets.

Scope of the Audit
The audit concerns a Solidity contract that defines the behavior of a Tokenized Fund.

The software is developed in the following repository:

```
https://github.com/karpatkey/karpatkey-tokenized-fund/
```

Specifically, the audit concerns the following files from the `development` branch

```
contracts/src/fund/IkpkShares.sol
contracts/src/fund/kpkShares.sol
```

We audited commit `e1c2ecc0125c1262270e4661624e2c2c41bf69f6`

Methodology

The audit report has been compiled on the basis of the findings from different auditors (Jelle and Ben). The auditors work independently. Each of the auditors has several years of experience in developing smart contracts and web3 applications.

Liability

The audit is on a best-effort basis. In particular, the audit does not represent any guarantee that the software is free of bugs or other issues, nor is it an endorsement by Team Omega of any of the functionality implemented by the software.

Summary of findings

We found **two high severity issues** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **3 issues as "medium**" - these are issues we believe you should definitely address. In addition, **9** issues were classified as "low", and **14** issues were classified as "info" - we believe the code would improve if these issues were addressed as well.

All high and medium issues, and most of the lower and info issues, were resolved by the team.

Severity	Number of issues	Number of resolved issues
High	2	2
Medium	3	3
Low	9	7
Info	14	13

Resolution

Team Omega delivered a preliminary report on September 24, 2025. Kpk subsequently made a number of changes to address the issues mentioned in this report, which resulted in the following commit:

```
0e7940c6941f4a1325b3d94c34d50b5cabcccfea
```

We checked the changes and the new code and added our observations to the issues below.

Findings

General

G1. Openzeppelin dependency is not pinned to a release [medium] [resolved]

The git submodule `contracts/lib/openzeppelin-contracts-upgradeable` is pointing to commit `37bfda9eeb81cca848a9989118485a73d002eae4` from August 19, 2025.

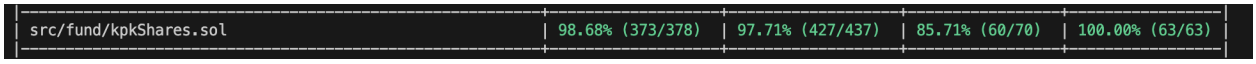
```
~/karpatkey-tokenized-fund (omega-audit)> git ls-tree HEAD -r | grep
openzeppelin-contracts-upgradeable
```

```
160000 commit 37bfda9eeb81cca848a9989118485a73d002eae4
contracts/lib/openzeppelin-contracts-upgradeable
```

This commit does not correspond to an official release by OpenZeppelin. The latest official release when writing this report is `5.4.0`, commit `e725abddf1e01cf05ace496e950fc8e243cc7cab`, from July 17.
There is no guarantee that pulling an non-release version from github is internally consistent or does not contain bugs. It is safer to work with an official release.
Recommendation: Use an official release.
Severity: Medium
Resolution: The issue was resolved as recommended: the dependency was set to commit `e725abddf1e01cf05ace496e950fc8e243cc7cab` which is release `5.4.0`

G2. Test coverage is incomplete [low] [not resolved]

The test coverage is overall high, but is not complete, and as evident from various issues in the report, multiple scenarios and edge cases have not been tested.



Recommendation: Increase the test coverage to %100 and create more comprehensive tests for possible edge cases.
Severity: Low
Resolution: The issue was not resolved. Test coverage has remained around the same.

kpkShares.sol

K1. Attacker can steal all assets and shares deposited in pending requests [high] [resolved]

The `cancelSubscription` and `cancelRedemption` functions, and the `updateSubscriptionRequest` and `updateRedemptionRequest` functions don't validate that the request passed to them is of the correct type.
Since those functions can issue refunds (cancel for the full amount, and update for any difference) based on the amounts in the requests, an attacker could abuse these functions to steal the assets deposits of other users in the contract.

An attack using the cancel functions to steal all funds that are currently in escrow in PENDING subscriptions would work as follows:

1. An attacker calls `requestRedemption` with 1 wei of share, the redemption asset being the asset to steal, and the share price inflated so that 1 wei of shares is “worth” the full amount in escrow. This will cost the attacker 1 wei of share.
2. The attacker will then call `cancelSubscription`, and pass it the ID of the *redemption* request that they just created. As there is no check on the type of request, this will pass, sending the attacker all the assets they asked for in their redemption requests - i.e. the entire balance.

An attacker could use essentially the same attack, with subscriptions and redemption inverted, to steal all user shares of all pending requests.
Another variant of the attack uses the update functions, which would allow the attacker to essentially perform the same attack, but without waiting for the TTL until cancellation to pass. This variant would also allow the attacker to cancel their own requests later to get back any of the small funds they deposited in their requests for the purpose of the attack.
Recommendation: Make sure to verify the type of request in the cancellation and update functions.
Severity: High
Resolution: This code was refactored and the issue is resolved.

K2. Update request can be used to steal funds and shares by front running approval [high] [resolved]

The `updateSubscriptionRequest` and `updateRedemptionRequest` functions allow users to update their subscription and redemption requests within a certain time period of `updateRequestTtl`. The issue is that if the operator approves a request before the `updateRequestTtl` passed, the user who submitted the request could front run the approval.

A scenario for this attack using redemption would be:

1. Create a redemption request
2. Wait for the operator to send a transaction in which the request is approved
3. Frontrun the operator’s transaction and call `updateRedemptionRequest` to steal as much as is approved from the portfolioSafe
4. The operator's transaction will now be mined using the updated data.

The same attack scenario can be used to steal any number of shares using a subscription request.
Recommendation: There are several types of solutions here. One solution could be to require that some time passes between the last update and the approval (e.g. that the `updateRequestTtl` of an order has passed before it can be approved). While this could work, it would mean each order would have to sit for (likely) at least a few days before it can be safely approved, meanwhile the shares and assets prices are very likely to change, and the order may

no longer be favorable for the user or the system. A better approach would be to follow the recommendation of K3 or K4, which will resolve this issue as well.
Severity: High
Resolution: The possibility of updating a request was removed, resolving the issue.

K3. Redemption requests are not executed at the stated price [medium] [resolved]

The current system architecture allows the users to pass the price of shares in USD as part of their requests. The system then consults an Oracle to get the price of the asset, which is then used to calculate how many assets corresponds to these shares. This amount is then stored as `assetAmount` on-chain, and logged in the `RedemptionRequest` event.
However, when the request is approved, this amount of assets a user gets is recalculated in the `_approveRedeemRequest` function using the Oracle's price at the time of approval (after reducing the fees), which may be different from the price when the request was created.

This seems unfair: the user has a good reason to expect that their order will be executed at (at least, see K4) the price when they created the order - especially because that is the price that is explicitly stated in the "receipt" (i.e. as stored on-chain and logged in the event).
Recommendation: One way around this is to let the user determine the price that they expect for their shares not in USD, but in terms of the assets that they will receive. This will give the user much more control over the price. This approach has the added benefit that there is no need anymore to consult an Oracle in the on-chain code. This makes sense, as *de facto* it is the operator that sets the price by choosing which requests to approve and which to reject. This would make the system considerably less complex and more secure.
Severity: Medium
Resolution: In the new version, the user specifies the minimum amount of shares/assets that they want to receive, and the order will be executed returning at least that amount minus the fees.

K4. Implementing limit orders would simplify architecture and give users better prices [medium] [resolved]

In the current implementation, requests for subscription (or redemption) are executed at the exact price that the user requests (although see issue K3). The result is that the requests in a batch that is approved by the operator will almost always be fulfilled with different prices, and those prices may all be different than the current live price at the time of execution.
This may be considered unfair, as prices may be significantly outdated or different between orders which are executed at the same time. A more fair system would take the user's price as a limit order - i.e. interpret the user's request not as a request to receive the *exact* number of shares (or assets) that they specify, but rather as a request to receive *at least* that amount.

The operator can then pass the price when calling `processSubscriptionRequests`, and execute orders at the price passed, while the code enforces the minimum amount the user requested to receive.
Recommendation: Allow the user to specify a minimum amount of assets/shares to receive, and have the operator set the price at execution. This change does not really represent a deviation from the current trust model, because already in the current implementation, the operator *de facto* sets the price of assets/shares when calling `processSubscriptionRequests` (or `processRedemptionRequests`) by choosing which requests to approve and which ones to reject.
Severity: Medium
Resolution: The issue was resolved as recommended

K5. Subscription and redemption requests do not expire [low] [resolved]

In the current system, users requests for subscription and redemption never expire, and can be executed at any time. A user can cancel their request after TTL has expired, but it may still be executed until the user actively calls to cancel it. Adding an explicit expiry could help users limit their exposure to price volatility by preventing execution after a user-specified period.
Recommendation: Implement an expiry date for requests - or expand the existing TTL logic.
Severity: Low
Resolution: The issue acknowledged, and the current behavior is a design choice. Users can cancel their requests after TTL.

K6. Oracle stale price check is too long [low] [resolved]

The `_getAssetPrice` function calls the oracle to get the asset price, and ensures the price is up to date by checking that it was updated in the last 24 hours. However, 24 hours is a lot of time: since most crypto markets can have significant volatility, this increases the risk of having a significant difference between the real price and the oracle price.
Recommendation: Reduce the stale price time limit, or have the operator provide the price as suggested in K4.
Severity: Low
Resolution: The dependency on oracles was removed

K7. Performance fee is based on first processed order share price [low] [resolved]

In `processSubscriptionRequests`, the performance fee is charged based on the share price of the first order passed in the `approveRequests`. This price may be different from the current price, or from other prices of other requests on the list, so this behaviour seems very arbitrary.

Recommendation: Pass the price to use for performance fee as part of the call so it always uses an up to date price. If you follow the recommendation of K4 you could use the price passed on processing.
Severity: Low
Resolution: All orders in a batch are now executed at the same price, and the fee is calculated on that price

K8. Malicious user can block the operator from stopping token deposits [low] [resolved]

The `_updateAsset` function has a check which reverts if the caller is trying to set `canDeposit` to `False` of an asset which has subscription requests.
A malicious user could submit 1 wei subscription requests to the system whenever the operator tries to stop a token deposit, and prevent the operator from completing the update.
Recommendation: There is no clear reason for the check, as pending orders with the suspended token can just be cancelled, so we recommend removing the check entirely.
Severity: Info
Resolution: The `_updateAsset` function was removed.

K9.TTLs may change during a request lifetime [low] [not resolved]

The `UserRequest` struct stores a timestamp which is used to calculate the TTL of a request and its update window. These TTLs are defined globally, and may change during a request lifetime. This could cause behaviour which is unexpected by the user, for example a user might expect to be able to cancel a request after a day, but then TTL may change and make the order impossible to cancel for a whole week.
Recommendation: Consider storing the `TTL` and `updateTTL` instead of the timestamp in the order struct, so TTLs won't change for existing orders.
Severity: Low
Resolution: The issue was acknowledged but not resolved.

K10. TTL regarding updates and cancellations may be implemented incorrectly [low] [resolved]

In the current implementation, the function updateSubscriptionRequest can only be executed within the timeframe given by `updateRequestTtl`, while the function cancelSubscription can only be executed after the timeframe given by `subscriptionRequestTtl`.
A similar logic is implemented for updating and cancelling redemptions.
Because cancelling a request is basically a special case of updating a request (a user can "cancel" a request by "updating" it with a price and amount that is almost-zero) - this basically means that either a request can *always* be canceled (if `subscriptionRequestTtl <`

`updateRequestTtl)` or, that a request can be canceled during the start and towards the end of its lifetime, but not in the middle. We assume this is a mistake.
Recommendation: Implement TTL consistently.
Severity: low
Resolution: As the update functions were removed, the inconsistency is not relevant anymore.

K11. Initialize does not emit events for initial values [low] [resolved]

The `initialize` function sets initial values for contract state variables, but does not emit corresponding update events, or an initialization event. This may make it harder or less efficient to cache the initial values in indexing services.
Recommendation: Emit update events for initial values. You may want to call the internal setters functions instead of setting the value directly on initialization.
Severity: Low
Resolution: The issue was resolved as recommended.

K12. Asset price of zero should revert [low] [resolved]

The `sharesToAssets` function checks if `normalizedPrice` is 0, and returns 0 in that case. However, if the price is 0, it is likely that an important issue has occurred with the price from the oracle, and returning a “good value” such as 0 may risk user funds. It is safer to revert in this case.
Recommendation: Revert if the price is 0 instead of returning 0.
Severity: Low
Resolution: The issue was resolved as recommended.

K13. Charge fees is called more than needed [info] [resolved]

In `processSubscriptionRequests`, the `_chargeFees` function is called for each approved request, despite the fact that the management and performance fees do not need to be called more than once every 24 hours (more than that is just wasting gas on checks), and the redemption fee is not charged on subscription requests at all.
Recommendation: If you want to couple the charge fees call with the process call, move it to be called once at the beginning (or end) of the function, and move the redemption fee charging to a separate function called for each redemption order on `_approveRedeemRequest`. Also consider passing the price from the operator and use that for the performance fee (see K4).
Severity: Info
Resolution: The issue was resolved as recommended.

K14. Impossible condition in _updateRedeemRequest [info] [resolved]

The `_updateRedeemRequest` has a condition to check if the `msg.sender` is not the investor of the request, and if so will try to spend allowance of the caller from the investor. However, this can never happen, since the function is only ever used in `updateRedemptionRequest`, which first calls `_validateRequestUpdate`, which requires the `msg.sender` to be the `investor`.
Recommendation: Remove the condition and call to spend allowance.
Severity: Info
Resolution: The update functions were removed from the code.

K15. Approved assets are stored twice but not updated in both locations [info] [resolved]

The `ApprovedAsset` struct is stored twice, both in `_approvedAssets` and `_approvedAssetsMap`. However, on `_updateAsset`, if the asset already exists, it does not get updated in `_approvedAssets`.
Recommendation: There is no reason to store `ApprovedAsset` twice. Instead, `_approvedAssets` can be an array of addresses, as that's the only thing it is really used for, and the `ApprovedAsset` struct should be saved only in `_approvedAssetsMap`.
Severity: Info
Resolution: The issue was resolved as recommended.

K16. Significant code duplication between the subscription and redemption requests [info] [resolved]

The `UserRequest` struct is designed as a single unified struct for both subscription and redemption requests, which should allow saving a lot of code duplication. However the current state of the code still contains significant code duplication separating the two flows, which could be reduced by moving common code to common functions.
Recommendation: Dedeplicate code that is shared in subscription and redemption logic.
Severity: Info
Resolution: The relevant code was completely refactored.

K17. Request ID is not needed as part of the UserRequest struct [info] [resolved]

The ID of a request is saved as part of its struct, but that struct property is never used, as you have to know that ID to even access the request in the mapping. So there is no reason to store it inside the request.
Recommendation: Remove the `requestId` from the `UserRequest` struct.

Severity: Info
Resolution: The issue was resolved as recommended.

K18. Requests are kept in storage beyond what is needed [info] [not resolved]

Requests are kept in storage even after processed or cancelled, and their status is updated accordingly. Yet there is no reason to keep them in storage, and gas could be saved by simply deleting a request once it's no longer pending.
Recommendation: Remove the status of requests and instead use an `isRequestActive` boolean to mark when a request is active. Then, once a request was processed or cancelled, simply delete the entire request.
Severity: Info
Resolution: This issue was not resolved.

K19. Functions should be marked as external when possible [info] [resolved]

The following functions are marked public but not used inside the contract and so should be marked external:

- `initialize`
- `requestRedemption`
- `processRedemptionRequests`
- `setSubscriptionRequestTtl`
- `setRedemptionRequestTtl`
- `setUpdateRequestTtl`
- `setFeeReceiver`
- `setManagementFeeRate`
- `setRedemptionFeeRate`
- `setPerformanceFeeRate`
- `setPerformanceFeeModule`
- `updateAsset`
- `getApprovedAsset`
- `getRequest`

Recommendation: Mark these functions as external.
Severity: Info
Resolution: The issue was resolved as recommended.

K20. Max fee check has wrong comment [info] [resolved]

The maximum fee rate checks for setting redemption fee and management fee have a comment that the maximum fee is 10%, while in fact it is 20%.
Recommendation: Fix the comments.
Severity: Info
Resolution: The issue was resolved as recommended.

K21. Constants visibility should be marked explicitly [info] [resolved]

The `NAV_DECIMALS, SECONDS_PER_YEAR, MIN_TIME_ELAPSED,` and `OPERATOR` constants are missing explicit visibility declaration.
Recommendation: Mark these constants as public.
Severity: Info
Resolution: The issue was resolved as recommended.

K22. Unused constant NAV_DECIMALS [info] [resolved]

The `NAV_DECIMALS` constant is unused and can be removed.
Recommendation: Remove `NAV_DECIMALS` from the contract.
Severity: Info
Resolution: The issue was resolved as recommended.

K23. Unnecessary helper function _burnShares [info] [resolved]

The `_burnShares` function is only used once, and simply calls `_burn`. It would be cleaner to just call `_burn` like `_transfer` and `_mint` are called directly.
Recommendation: Remove `_burnShares` and use `_burn` directly instead.
Severity: Info
Resolution: The issue was resolved as recommended.

K24. Unnecessary use of struct for ApproveRedemptionRequest [info] [resolved]

The `ApproveRedemptionRequest` struct is unnecessary as it only has the request ID, and is used in redemption approval while for subscription approval the request ID is passed directly as a `uint256`.
Recommendation: Remove the unnecessary `ApproveRedemptionRequest` struct and pass uint256 IDs like in subscription approvals.
Severity: Info
Resolution: The issue was resolved as recommended.

K25. Duplicated check in requestSubscription and requestRedemption [info] [resolved]

The `requestSubscription` and `requestRedemption` functions check if the asset specified can be deposited/ redeemed according to the asset config and revert if not. Yet this is also checked in the `previewSubscription` and `previewRedemption` functions which are called before that, meaning the check in the request functions can never revert and is duplicated.
Recommendation: Remove the check of canDeposit and canRedeem from `requestSubscription` and `requestRedemption`.
Severity: Info
Resolution: The issue was resolved as recommended.

K26. Unused event parameter in SubscriptionRequest and RedemptionRequest [info] [resolved]

The `SubscriptionRequest` and `RedemptionRequest` events have a supply parameter which is not used and can be removed.
Recommendation: Remove the supply parameter of the `SubscriptionRequest` and `RedemptionRequest` events.
Severity: Info
Resolution: The issue was resolved as recommended.