Karak Restaking Audit Report

Version 2.0

Audited by:

HollaDieWaldfee

bytes032

alexxander

April 1, 2024

Contents

1 Introduction

1.1 About Renascence

Renascence Labs was established by a team of experts including HollaDieWaldfee, MiloTruck, alexxander and bytes032.

Our founders have a distinguished history of achieving top honors in competitive audit contests, enhancing the security of leading protocols such as Reserve Protocol, Arbitrum, MaiaDAO, Chainlink, Dodo, Lens Protocol, Wenwin, PartyDAO, Lukso, Perennial Finance, Mute and Taurus.

We strive to deliver tailored solutions by thoroughly understanding each client's unique challenges and requirements. Our approach goes beyond addressing immediate security concerns; we are dedicated to fostering the enduring success and growth of our partners.

More of our work can be found here.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an 'as-is' and 'as-available' basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

1.3.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol’s core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol’s functionality
- Low - Funds are **not** at risk

1.3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

2 Executive Summary

2.1 About Karak Restaking

Karak-Restaking is still under active development. Current features in the protocol are:

- ERC4626 Vaults in which users can stake that also support Permit deposits.
- Vault Supervisor contract that serves as the staking entry point and also orchestrates the Vaults `deposit`, `withdraw`, `mint` and `redeem` operations. The Vault Supervisor is also responsible for creating and adding new Vaults to the system.
- Delegate Supervisor contract that handles withdrawals by first queuing the withdraw requests and then executing them if the user has exceeded the staking lock period for each Vault he has requested to exit.

2.2 Overview

Project	Karak Restaking
Repository	karak-restaking
Commit Hash	46b15e107cb6...
Mitigation Hash	0f6461f2d63d...
Date	25 March 2024 - 27 March 2024

2.3 Issues Found

Severity	**Count**
High Risk	0
Medium Risk	0
Low Risk	4
Informational	7
Total Issues	**11**

3 Findings Summary

ID	Description	Status
L-1	Deposits through `VaultSupervisor.depositWithSignature()` can be griefed	Resolved
L-2	Effective delay in withdrawal request is maximum of vault delays	Acknowledged
L-3	Vault should override `_underlyingDecimals()` from Solady	Resolved
L-4	`MAX_WITHDRAWAL_DELAY` constant depends on block time of different chains	Resolved
I-1	`DelegationSupervisor.startWithdraw()` can create a `WithdrawRequest` with empty `vaults` and `shares` arrays	Resolved
I-2	Functions `setDelegationSupervisor()` and `modifyVaultAllowlist()` inside `VaultSupervisor` might brick pending withdrawals	Resolved
I-3	Vault withdrawal delay cannot be changed and is not checked upon initialization	Resolved
I-4	Improvements in tests	Resolved
I-5	Upgradeable contracts `Vault`, `VaultSupervisor` and `DelegationSupervisor` are missing a call to `_disableInitializers()`	Resolved
I-6	Interface declaration `IDelegationSupervisor.initialize()` has wrong parameter names	Resolved
I-7	Withdrawal incentives are broken once rewards are distributed as Vault yield	Acknowledged

4 Findings

Low Risk

[L-1] Deposits through `VaultSupervisor.depositWithSignature()` can be griefed

Context:

- VaultSupervisor

Description: The `VaultSupervisor.depositWithSignature()` function has two front-running issues.

```
## VaultSupervisor.sol

function depositWithSignature(
    IVault vault,
    uint256 amount,
    address user,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external nonReentrant whenNotPaused returns (uint256 shares) {
    IERC20Permit(address(vault.asset())).permit(user, address(vault), value,
    deadline, v, r, s);
    return depositInternal(user, vault, amount);
}
```

1. An attacker can front-run the function call and specify a `amount` parameter that is lower than `value`. It can be any value in the range `[x,value]`, such that `x` translates to a non-zero `shares` amount in the `Vault`. This is effectively a DoS since the user doesn't deposit the intended amount.
2. An attacker can extract the permit signature from the transaction and execute the permit directly on the ERC20 token. The transaction to `depositWithSignature()` will revert.

Recommendation: Two observations lead to the recommendation.

1. A failed call to `ERC20Permit.permit()` must not cause a revert to `depositWithSignature()`. Therefore, the call to `ERC20Permit.permit()` must be wrapped in a `try-catch` block.
2. Without requiring a successful call to `ERC20Permit.permit()`, the `depositWithSignature()` function lacks an authorization check. Any allowance that a user has given to the `Vault` could be used to deposit his funds. This leads to the necessity of using a second signature to specify the user's intent to deposit.

In summary, the logic can be described by the following pseudocode:

```
function depositWithSignature(
    depositSignature
    permitSignature
) {
    try ERC20Permit.permit(permitSignature) catch {}
    checkSignature(depositSignature)
    depositInternal(depositSignature.user, depositSignature.vault,
    depositSignature.amount)
}
```

Karak: Fixed in PR22 and PR34.

Renascence: The recommendation has been implemented.

[L-2] Effective delay in withdrawal request is maximum of vault delays

Context:

- Withdraw.sol

Description: Currently, `Withdraw.finishStartedWithdrawal()` loops over the Vaults that are recorded in the `QueuedWithdrawal` request, and for each Vault checks if the per Vault delay `vaultWithdrawalDelay` has passed. The problem is that this mechanism sets the effective withdrawal delay for the whole `QueuedWithdrawal` request to the maximum Vault delay among the Vaults in the request. This could impact the user experience since a user would need to wait for the maximum delay to pass before he can withdraw even from a Vault with a much shorter delay.

```
## Withdraw.sol

function finishStartedWithdrawal(QueuedWithdrawal calldata withdrawal,
DelegationSupervisorLib.Storage storage delegationSupervisor) external {
    ...

    for (uint256 i = 0; i < withdrawal.request.vaults.length;) {
        if (withdrawal.start +
delegationSupervisor.state.vaultWithdrawalDelay[withdrawal.request.vaults[i]] >
block.timestamp) {
            revert MinWithdrawDelayNotPassed();
        }
        ...
    }
}
```

Recommendation: If this behavior is determined to be correct, the finding can be acknowledged. The user can queue multiple `WithdrawRequests` through `DelegationSupervisor.startWithdraw()` and avoid the issue, however, the user will be required to spend excess gas. If the behavior is not expected, a partial withdrawal mechanism should be implemented, such that for each Vault the partial withdrawal can be processed as soon as the delay for the Vault has passed.

Karak: Acknowledged.

Renascence: The finding has been acknowledged.

[L-3] Vault should override `_underlyingDecimals()` from Solady

Context:

- Vault.sol

Description: The `Vault` contract should override Solady's `ERC4626._underlyingDecimals()` in case the decimals of the underlying assets are not the default 18.

Recommendation: The Solady library suggests using `ERC4626._tryGetAssetDecimals()` during initialization to set the decimals of the underlying asset.

```diff
@@ -19,6 +19,8 @@ contract Vault is ERC4626, Initializable, Ownable,
PausableUpgradeable, Reentran
     string private symbolStr;
     uint256[45] private __gap;

+    uint8 decimals;

     /* ========== MUTATIVE FUNCTIONS ========== */

     function initialize(IVaultSupervisor _supervisor, IERC20 _depositToken, string
     memory _name, string memory _symbol)
@@ -31,6 +33,13 @@ contract Vault is ERC4626, Initializable, Ownable,
PausableUpgradeable, Reentran
         depositToken = _depositToken;
         nameStr = _name;
         symbolStr = _symbol;
+
+        (bool success, uint8 result) = _tryGetAssetDecimals(address(_depositToken));
+        decimals = success ? result : _DEFAULT_UNDERLYING_DECIMALS;
+    }
+
+    function _underlyingDecimals() internal view override returns (uint8) {
+        return decimals;
     }
```

Karak: Fixed in PR24.

Renascence: The recommendation has been implemented.

[L-4] `MAX_WITHDRAWAL_DELAY` constant depends on block time of different chains

Context:

- Constants.sol

Description: The `MAX_WITHDRAWAL_DELAY` constant is set to `216000 * 12`, where `12` represents the number of seconds per block on Ethereum. However, as discussed with the client, the protocol will be deployed on Ethereum and Karak, and 12 seconds is an incorrect block time on Karak.

Recommendation: It is recommended to make `MAX_WITHDRAWAL_DELAY` independent from the block time of the chain that the protocol is deployed on. The constant should be set to the intended number of seconds which will be correct on all chains.

Karak: Fixed in PR26.

Renascence: `MAX_WITHDRAWAL_DELAY` is set to 30 days, this is equivalent to `216000 * 12`, which it has been set to before. The value of 30 days indicates that it is intended to be the same value across chains.

Informational

[I-1] `DelegationSupervisor.startWithdraw()` can create a `WithdrawRequest` with empty `vaults` and `shares` arrays

Context:

- DelegationSupervisor

Description: Currently, the functions `DelegationSupervisor.startWithdraw()` and `DelegationSupervisor.removeSharesAndStartWithdrawal()` are missing validation checks to ensure that the processed `WithdrawRequest` doesn't contain empty `vaults` and `shares` arrays.

```
# DelegationSupervisor

function startWithdraw(Withdraw.WithdrawRequest[] calldata withdrawalRequests)
    ...

    for (uint256 i = 0; i < withdrawalRequests.length;) {
        // @audit no check that vaults and shares are non-empty
        if (withdrawalRequests[i].vaults.length !=
withdrawalRequests[i].shares.length) {
            revert InvalidInput();
        }
        if (withdrawalRequests[i].withdrawer != msg.sender) {
            revert NotStaker();
        }
        // Remove shares from staker's strategies and place strategies/shares in
queue.
        (withdrawalRoots[i], withdrawConfigs[i]) = removeSharesAndStartWithdrawal({
            ...
    }
}
```

```
# DelegationSupervisor

function removeSharesAndStartWithdrawal(
    ...
    IVault[] memory vaults,
    uint256[] memory shares
) internal returns (bytes32 withdrawalRoot, Withdraw.QueuedWithdrawal memory
withdrawal) {
    // @audit vaults and shares can be empty
    if (vaults.length != shares.length) revert InvalidInput();
    //|| operator == address(0)
    if (staker == address(0)) revert ZeroAddress();
    if (staker != withdrawer) revert NotStaker();

    for (uint256 i = 0; i < vaults.length;) {
        //_decreaseOperatorShares(operator, vaults[i], shares[i]);
        self.config.vaultSupervisor.removeShares(staker, vaults[i], shares[i]);
        unchecked {
            i++;
        }
    }
    ...
}
```

Recommendation: This finding can currently be acknowledged since it doesn't lead to any particular impact. However, we advise that 0 length checks for `vaults` and `shares` are implemented in order to improve the robustness of the code against malformed input.

Karak: Fixed in PR27.

Renascence: The checks that `vaults` and `shares` are non-empty, is performed before a withdrawal request is created in `Withdraw.validate()`.

[I-2] Functions `setDelegationSupervisor()` and `modifyVaultAllowlist()` inside `VaultSupervisor` might brick pending withdrawals

Context:

- VaultSupervisor

Description: Currently, there is no migration process for replacing Delegation Supervisors. This means using `VaultSupervisor.setDelegationSupervisor()` will brick all pending withdrawals that were started with the old supervisor, since the function `VaultSupervisor.redeemShares()` has the `onlyDelegationSupervisor` modifier, which will allow calls only from the new Delegation Supervisor. Similarly, if a user has a queued withdrawal that contains a Vault that has been disallowed through `VaultSupervisor.modifyVaultAllowlist()`, he will loose on all of his pending withdrawals that are from the allowed Vaults (and are within the pending withdraw request).

Recommendation: This finding can be acknowledged since the owner is fully trusted and is expected to carefully perform privileged actions.

Nevertheless, a mapping could be introduced that keeps track of previous Delegation Supervisors and the mapping would be used by a modifier to allow older Delegation Supervisors to call `VaultSupervisor.redeemShares()`.

As the project will be extended with new features, the development roadmap should be taken into consideration when deciding how to address this finding. The recommended solution with the mapping may conflict with other features.

Karak: Fixed in PR31.

Renascence: The recommendation has been implemented.

[I-3] Vault withdrawal delay cannot be changed and is not checked upon initialization

Context:

- DelegationSupervisorLib

Description: There are two delays that are enforced in the `Withdraw.finishStartedWithdrawal()` function.

- The `delegationSupervisor.config.minWithdrawalDelay`, which is required for all withdrawals
- The `delegationSupervisor.state.vaultWithdrawalDelay[vaultId]` which is a withdrawal delay set per Vault

There are two minor issues with how vault delays are currently set up.

- Both delays cannot be changed once initialized unless the contracts are upgraded.
- The per Vault delay is checked not to exceed `Constants.MAX_WITHDRAWAL_DELAY` in `DelegationSupervisorLib.setMinWithdrawOfVaults()`, however, the general `minWithdrawalDelay` is not checked if it exceeds `Constants.MAX_WITHDRAWAL_DELAY` and can be set to any value in `DelegationSupervisorLib.initOrUpdate()`.

Recommendation: This finding can be acknowledged since it's a design decision and the owner is trusted to set up a correct `minWithdrawalDelay`. However, for the sake of maintaining a good code standard, `delegationSupervisor.config.minWithdrawalDelay` could be checked against a constant similar to how the per Vault delay is checked not to exceed `MAX_WITHDRAWAL_DELAY`.

Karak: Fixed in PR31.

Renascence: `minWithdrawalDelay` is now checked in `DelegationSupervisorLib.initOrUpdate()`. Also, all withdrawal delays can now be updated by the owner.

[I-4] Improvements in tests

Context:

- DelegationSupervisor.t.sol
- Vault.t.sol
- VaultSupervisor.t.sol

Description: The `Vault`, 'VaultSupervisor', and `DelegationSupervisor` contracts are supposed to be upgradeable and deployed behind `Proxy` contracts. The current test suite and script folder of the project does not include tests and scripts that mimic how the contracts will be deployed in practice.

Recommendation: Extend the test suite to include `Proxy` tests and add a deployment script.

Karak: Fixed in PR32.

Renascence: The recommendation has been implemented.

[I-5] Upgradeable contracts `Vault`, `VaultSupervisor` and `DelegationSupervisor` are missing a call to `_disableInitializers()`

Context:

- DelegationSupervisor
- Vault
- VaultSupervisor

Description: The best practice in contracts that inherit from `Initializable` is to disable the initializers since if left uninitialized they can be invoked in the implementation contract by an attacker. For example, there is a past vulnerability disclosure that demonstrates how initializers getting called in the implementation can lead to contract takeover where the attacker can appoint an owner and would self-destruct the implementation, therefore, bricking the Proxy: OZ post-mortem. Although this issue has been fixed from OZ version 4.3.2 it's still best practice to call `Initializable._disableInitializers()` in a constructor in the implementation.

```
# Initializable.sol

* [CAUTION]
 * ====
 * Avoid leaving a contract uninitialized.
 *
 * An uninitialized contract can be taken over by an attacker. This applies to both a
 proxy and its implementation
 * contract, which may impact the proxy. To prevent the implementation contract from
 being used, you should invoke
 * the {_disableInitializers} function in the constructor to automatically lock it
 when it is deployed:
 *
```

Recommendation: Add a constructor with a call to `_disableInitializers()` in the `Vault`, `VaultSupervisor`, and `DelegationSupervisor` contracts.

```
+       constructor() {
+           _disableInitializers();
+       }
```

Karak: Fixed in PR29.

Renascence: The recommendation has been implemented.

[I-6] Interface declaration `IDelegationSupervisor.initialize()` has wrong parameter names

Context:

- IDelegationSupervisor.sol

Description: The declaration of `IDelegationSupervisor.initialize()` has parameters called `_minWithdrawDelayBlocks` and `_withdrawalDelayBlocks` that suggests delays are recorded in `blocks`, however, the delays are supposed to be in seconds.

Recommendation: Change the variables to `minWithdrawDelay` and `withdrawalDelays`, similar to how it is in the implementation `DelegationSupervisor.initialize()`.

Karak: Fixed in PR15.

Renascence: The recommendation has been implemented.

[I-7] Withdrawal incentives are broken once rewards are distributed as Vault yield

Context:

- DelegationSupervisor.sol

Description: The protocol enforces a withdrawal delay period for each `Vault`. Withdrawals from the `Vault` need to be queued and can only be finished when the withdrawal delay has passed.

It has been discovered that shares are only redeemed upon finishing the withdrawal. Hence, shares continue to earn yield during the withdrawal delay period.

Users can bypass the withdrawal delay period by preemptively queuing withdrawals, such that they can withdraw instantly when they want to.

This effectively caps the duration for which the withdrawal delay is applicable to one such withdrawal delay starting at the deposit time, since if the preemptive withdrawal is started immediately at the deposited time, withdrawals are instant after one withdrawal delay period has passed.

Currently, there does not exist an issue since rewards are distributed as ”points” which are calculated off-chain and they won't be rewarded for any shares that are queued for withdrawal.

Recommendation: The finding should be tracked internally and a mitigation must be implemented as soon as the protocol switches from ”points” to Vault yield.

Karak: Acknowledged. In the future, the startWithdraw TX would ”sell” the shares.

Renascence: Acknowledged, as recommended.

5 Centralization Risks

5.1 Owner is fully trusted

All contracts will be deployed behind proxies which means they can be changed to execute arbitrary logic. In addition, the contracts contain prviliged functions that the owner can call. In summary, the owner and proxy admin must be fully trusted.

6 Systemic Risks

6.1 External tokens risk

Users participate in the protocol by depositing underlying tokens in the `Vaults`. The security of the `Vaults` relies on the security of the underlying tokens. If the underlying tokens lose their value or get hacked, this is an immediate loss of funds for the users.