RENASCENCE

# Karak Audit Report

Version 2.0

Audited by:

**MiloTruck**

**alexxander**

**marchev**

July 16, 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

Karak enables users to repurpose their staked assets to other applications. Stakers can allocate their assets to a Distributed Secure Service (DSS) on the Karak network and agree to grant additional enforcement rights to their staked assets.

The opt-in feature creates additional slashing conditions to meet the conditions of secured services such as data availability protocols, bridges, or oracles.

## 2.2 Overview

| | |
|---|---|
| Project | Karak |
| Repository | karak-restaking |
| Commit Hash | 8aad9ad5592e... |
| Mitigation Hash | d928a0803b62... |
| Date | 2 July 2024 - 6 July 2024 |

## 2.3 Issues Found

| **Severity** | **Count** |
|---|---|
| High Risk | 10 |
| Medium Risk | 7 |
| Low Risk | 5 |
| Informational | 6 |
| **Total Issues** | **28** |

# 3 Findings Summary

| ID | Description | Status |
|---|---|---|
| H-1 | Unsafe cast from `int256` to `uint256` in `NativeVault._updateBalance()` will overflow | Resolved |
| H-2 | Increasing `self.totalAssets` before share calculation in `NativeVault._increaseBalance()` mints less shares | Resolved |
| H-3 | Missing increment for `node.activeValidatorCount` in `NativeVault.validateWithdrawalCredentials()` | Resolved |
| H-4 | `NativeVault.validateExpiredSnapshot()` cannot be called on a node owner with no active validators | Resolved |
| H-5 | `NativeVault._startSnapshot()` reverts with an arithmetic underflow when a native node's balance decreases | Resolved |
| H-6 | `assets > withdrawableWei()` check in `_decreaseBalance()` could DOS `NativeVault._updateSnapshot()` | Resolved |
| H-7 | Calling `validateWithdrawalCredentials()` followed by `startSnapshot()/validateExpiredSnapshot()` will permanently DOS snapshots | Resolved |
| H-8 | Specifying `validatorIndex` as `uint64` allows `BeaconProofsLib.validateValidatorProof()` to pass with incorrect proofs | Resolved |
| H-9 | Wrong withdraw address verification in `NativeVaultLib.validateWithdrawalCredentials()` | Resolved |
| H-10 | Wrong use of the beacon block root instead of the beacon state root in `NativeVault.validateWithdrawalCredentials()`. | Resolved |
| M-1 | `NativeVault.finishWithdrawal()` doesn't reset `withdrawalMap[withdrawalKey]` after executing the withdrawal | Resolved |
| M-2 | `Native.startWithdrawal()` can be called repeatedly to queue an infinite number of withdrawals | Resolved |
| M-3 | `assets > withdrawableWei()` check in `_decreaseBalance()` causes `NativeVault.finishWithdrawal()` to revert when slashing occurs | Resolved |
| M-4 | Missing `validatorProof.length` check in `BeaconProofsLib.validateValidatorProof()` | Resolved |
| M-5 | Function `_getParentBlockRoot` limits the beacon roots lookback window | Resolved |
| M-6 | `NativeVaultLib.validateWithdrawalCredentials()` should return the actual balance of the validator | Acknowledged |
| M-7 | Inability to update node implementation in `NativeVault` | Resolved |
| L-1 | `NativeVault.slashAssets()` could incorrectly return `0` | Resolved |
| L-2 | A node owner's `totalRestakedETH` and shares will lag behind his actual restaked balance | Resolved |

| ID | Description | Status |
|---|---|---|
| L-3 | NativeVault.startWithdrawal() should ensure the node owner's last snapshot has not expired | Resolved |
| L-4 | The operator of NativeVault is not stored in NativeVaultLib.Storage | Resolved |
| L-5 | Redundant pausing modifier in NativeVault.createNode() | Resolved |
| I-1 | NativeVault.validateWithdrawalCredentials() should call _increaseBalance() to avoid an unsafe int256 cast | Resolved |
| I-2 | STATE_ROOT_IDX and BEACON_STATE_ROOT_IDX can be combined in a single constant | Resolved |
| I-3 | Unused NativeNode pause functions by NativeVault | Acknowledged |
| I-4 | Reverting receive() function in NativeNode is redundant | Resolved |
| I-5 | Missing reentrancy guards for functions in NativeVault, NativeNode and SlashStore | Resolved |
| I-6 | Code improvements | Resolved |

# 4 Findings

## High Risk

### [H-1] Unsafe cast from `int256` to `uint256` in `NativeVault._updateBalance()` will overflow

**Context:** NativeVault.sol#L482-L487

**Description:** In `NativeVault._updateBalance()`, `assets` is cast from `int256` to `uint256` directly as such:

```
function _updateBalance(address _of, int256 assets) internal {
    if (assets > 0) {
        _increaseBalance(_of, uint256(assets));
    } else if (assets < 0) {
        _decreaseBalance(_of, uint256(assets));
    } else {
```

However, if `assets` is negative, casting it to `uint256` directly will cause `assets` to overflow into a huge value. This will cause `_updateBalance()` to revert whenever it is called to decrease a user's balance, making it impossible to withdraw from the protocol.

**Recommendation:** Multiply `assets` by `-1` first before casting to `uint256`:

```
  } else if (assets < 0) {
-     _decreaseBalance(_of, uint256(assets));
+     _decreaseBalance(_of, uint256(-assets));
  } else {
```

Note that this method of converting `int256` to `uint256` does not work if `assets` happens to be `type(int256).min`. However, `assets` should never reach that value under normal conditions.

**Karak:** Fixed in PR 341.

**Renascence:** Verified, the recommended fix was implemented.

### [H-2] Increasing `self.totalAssets` before share calculation in `NativeVault._increaseBalance()` mints less shares

**Context:** NativeVault.sol#L464-L466

**Description:** In `NativeVault._increaseBalance()`, `self.totalAssets` is increased by `assets` before calculating the amount of shares to mint to the receiver:

```
self.totalAssets += assets;
uint256 shares = convertToShares(assets);
_mint(_of, shares);
```

However, increasing `self.totalAssets` first causes `convertToShares()` to calculate less shares to be minted than expected, since `totalAssets()` is increased beforehand.

A naive example:

- Assume the following:
  - `self.totalAssets = 100e18`
  - `totalSupply = 100e18`
- `_increaseBalance()` is called with `assets = 100e18`:
  - `self.totalAssets = 100e18 + 100e18 = 200e18`
- `convertToShares()` calculates `shares` as `50e18` as:

```
assets * (totalSupply + 1) / (totalAssets + 1) = 100e18 * (100e18 + 1) / (200e18 + 1)
= 50e18
```

- However, `50e18` shares is only worth roughly `66.6e18` of assets as:

```
shares * (totalAssets + 1) / (totalSupply + 1) = 50e18 * (200e18 + 1) / (150e18 + 1)
= ~66.6e18
```

In the example above, the user loses around `33.3e18` assets.

**Recommendation:** The correct order of operator would be to increase `self.totalAssets` after calling `convertToShares()`:

```
- self.totalAssets += assets;
  uint256 shares = convertToShares(assets);
  _mint(_of, shares);
+ self.totalAssets += assets;
```

**Karak:** Fixed in PR 325.

**Renascence:** Verified, the recommended fix was implemented.

### [H-3] Missing increment for `node.activeValidatorCount` in `NativeVault.validateWithdrawalCredentials()`

**Context:** NativeVault.sol#L182-L189

**Description:** Node owners call `NativeVault.validateWithdrawalCredentials()` to add active validators to their native node:

```
for (uint256 i = 0; i < validatorFieldsProofs.length; i++) {
    totalRestakedWei += self.validateWithdrawalCredentials(
        nodeOwner,
        beaconStateRootProof.timestamp,
        _getParentBlockRoot(beaconStateRootProof.timestamp),
        validatorFieldsProofs[i]
    );
}
```

However, after calling `NativeVaultLib.validateWithdrawalCredentials()` to add all active validators in the loop above, the function does not increment `node.activeValidatorCount` (ie. the number of active validators in a native node) by the number of new validators added.

This makes it impossible to update the validator's balance in future snapshots as the number of active validators for all native nodes will always remain at `0`.

**Recommendation:** Increment `node.activeValidatorCount` by the number of active validators added as such:

```
  for (uint256 i = 0; i < validatorFieldsProofs.length; i++) {
      totalRestakedWei += self.validateWithdrawalCredentials(
          // ...
      );
  }
+ node.activeValidatorCount += validatorFieldsProofs.length;
```

**Karak:** Fixed in PR 323.

**Renascence:** Verified, the issue was fixed by incrementing `activeValidatorCount` in `NativeVaultLib.validateWithdrawalCredentials()` each time it is called.

### [H-4] `NativeVault.validateExpiredSnapshot()` cannot be called on a node owner with no active validators

**Context:** NativeVault.sol#L210-L215

**Description:** `NativeVault.validateExpiredSnapshot()` contains the following checks:

```
NativeVaultLib.ValidatorDetails memory validatorDetails =
node.validatorPubkeyHashToDetails[validatorPubkey];

if (beaconStateRootProof.timestamp < validatorDetails.lastBalanceUpdateTimestamp +
Constants.SNAPSHOT_EXPIRY) {
    revert SnapshotNotExpired();
}
if (validatorDetails.status != NativeVaultLib.ValidatorStatus.ACTIVE) revert
ValidatorNotActive();
```

As seen from above, `validateExpiredSnapshot()` can only be called when an active validator's `lastBalanceUpdateTimestamp` is more than 7 days ago. As such, it is not possible to call `validateExpiredSnapshot()` when a user has no active validators, even if his last snapshot has expired.

When slashing occurs, this would make it impossible to forcefully update a node owner's snapshot. As a result, the node owner's balance will never be updated and `slashStore` might never receive the slashed funds.

For example:

- Node owner has one active validator with 32 ETH.
- Node owner performs a full withdrawal for his validator. Its status is now `WITHDRAW` and the 32 ETH is moved into his native node.
- Karak operator calls `slashAssets()` to perform slashing, which reduces his balance to 31 ETH.
- Since the node owner has no more active validators, `validateExpiredSnapshot()` cannot be called.

In this scenario, it is impossible to forcefully move 1 ETH from the node owner's native node into `slashStore` and update his balance. If the node owner chooses to withdraw his remaining 31 ETH and never calls `startSnapshot()`, `slashStore` will never receive the 1 ETH that was slashed.

**Recomendation:**

Consider checking if a node owner's last snapshot has expired with `node.lastSnapshotTimestamp` instead:

```
function validateExpiredSnapshot(
    address nodeOwner,
) external nodeExists(nodeOwner)
whenFunctionNotPaused(Constants.PAUSE_NATIVEVAULT_VALIDATE_EXPIRED_SNAPSHOT) {
    NativeVaultLib.Storage storage self = _state();
    NativeVaultLib.NativeNode storage node = self.ownerToNode[nodeOwner];

    if (block.timestamp < node.lastSnapshotTimestamp + Constants.SNAPSHOT_EXPIRY) {
        revert SnapshotNotExpired();
    }

    _startSnapshot(node, false, nodeOwner);
}
```

**Karak:** Fixed in PR 335. The earlier design was supposed to check if the validator has been slashed on the beacon chain as well. However, based on this issue we realized introducing that check would complicate our flow and just allowing anyone to start a snapshot after an expiry period seemed the better way to go.

**Renascence:** Verified, the recommended fix was implemented.

### [H-5] `NativeVault._startSnapshot()` reverts with an arithmetic underflow when a native nodes balance decreases

**Context:**

- NativeVault.sol#L448
- NativeVault.sol#L422-L423

**Description:** `node.creditedNodeETH` stores the cumulative amount of ETH ever held by the native node as it is increased in `_updateSnapshot()` by `nodeBalanceWei`:

```
node.creditedNodeETH += snapshot.nodeBalanceWei;
```

Note that `node.creditedNodeETH` is not modified anywhere else in the code.

`node.creditedNodeETH` is used in `_startSnapshot()` to calculate the amount of ETH gained by the native node since the last snapshot:

```
// Calculate unattributed node balance
uint256 nodeBalanceWei = node.nodeAddress.balance - node.creditedNodeETH;
```

However, when the native node transfers ETH out, its ETH balance will become smaller than `node.creditedNodeETH`. Afterwards, when `_startSnapshot()` is called, `node.nodeAddress.balance - node.creditedNodeETH` will revert with an underflow.

For example:

- Assume a native node holds 2 ETH. Both nodeAddress.balance and creditedNodeETH are 2e18.
- The node owner withdraws 1 ETH, which transfers 1 ETH out from the native node.
- When _startSnapshot() is called afterwards:
  - nodeAddress.balance - creditedNodeETH = 1e18 - 2e18, which reverts with an underflow.

This makes it impossible for the node owner's snapshot to ever be updated. As such, his number of shares will never increase even if his total restaked balance increases from ETH rewards.

**Recommendation:** creditedNodeETH should store the native node's ETH balance during the last snapshot.

In _updateSnapshot(), consider removing the line adding nodeBalanceWei to creditedNodeETH:

```
- node.creditedNodeETH += snapshot.nodeBalanceWei;
```

Instead, set creditedNodeETH to the node's current balance in _startSnapshot(). Additionally, nodeBalanceWei should be 0 when the native node's ETH balance decreases:

```
  // Calculate unattributed node balance
- uint256 nodeBalanceWei = node.nodeAddress.balance - node.creditedNodeETH;
+ uint256 nodeBalanceWei;
+ if (node.nodeAddress.balance > node.creditedNodeETH) {
+   nodeBalanceWei = node.nodeAddress.balance - node.creditedNodeETH;
+ }
+ node.creditedNodeETH = node.nodeAddress.balance;
```

This ensures nodeBalanceWei will always be the amount of ETH received by the native node after the last snapshot.

**Karak:** Fixed in PR 330.

**Renascence:** Verified, the issue has been fixed by subtracting from node.creditedNodeETH on withdrawal and reducing it to the balance of the native node in _transferToSlashStore().

### [H-6] `assets > withdrawableWei()` check in `_decreaseBalance()` could DOS `NativeVault._updateSnapshot()`

**Context:**

- NativeVault.sol#L454
- NativeVault.sol#L473

**Description:** Whenever a snapshot is completed, `NativeVault._updateSnapshot()` calls to update the node owner's balance:

```
_updateBalance(nodeOwner, totalDeltaWei);
```

If `totalDeltaWei` happens to be negative, `_updateBalance()` calls `_decreaseBalance()`, which checks that `totalDeltaWei` is not greater than `withdrawableWei()`:

```
if (assets > withdrawableWei(_of)) revert WithdrawMoreThanMax();
```

Note that `withdrawableWei()` returns the minimum between the node owner's native node balance and the assets equivalent of his shares.

However, this check could cause `_updateSnapshot()` to incorrectly revert when completing a snapshot. For example:

- Assume a node owner has 32 ETH in a validator and no ETH in his native node.
- The following events occur:
  - His native node receives 0.3 ETH from validator rewards.
  - The beacon chain slashes his validator for 1 ETH, leaving 31 ETH remaining.
- He calls `startSnapshot()`, which sets `nodeBalanceWei = 0.3 ether` as his native node gained 0.3 ETH.
- He calls `validateSnapshotProofs()`, which sets `balanceDeltaWei = -1 ether` as his validator lost 1 ETH.
- When `_updateSnapshot()` is called:
  - `totalDeltaWei = 0.3 ether - 1 ether = -0.7 ether`
  - `_decreaseBalance()` is called with `assets = 0.7 ether`.
  - `withdrawableWei()` returns his native node's balance, which is 0.3 ETH.
  - Since `assets > withdrawableWei()`, the function reverts.

As seen from above, if a node owner's validators are slashed for more than his native node's current balance, `_updateSnapshot()` will always revert when called. This makes it impossible to update his snapshot, even after it expires.

**Recommendation:** Consider removing the `assets > withdrawableWei(_of)` check from `_decreaseBalance()`:

```
  function _decreaseBalance(address _of, uint256 assets) internal {
      NativeVaultLib.Storage storage self = _state();
-     if (assets > withdrawableWei(_of)) revert WithdrawMoreThanMax();
```

This check should be moved into `finishWithdrawal()` instead to ensure the user cannot withdraw assets than he should be able to.

**Karak:** Fixed in PR 342.

**Renascence:** Verified, the check was removed from `_decreaseBalance()`.

### [H-7] Calling `validateWithdrawalCredentials()` followed by `startSnapshot()`/`validateExpiredSnapshot()` will permanently DOS snapshots

**Context:**

- NativeVault.sol#L172-L175
- NativeVaultLib.sol#L179
- NativeVault.sol#L145-L147

**Description:** `NativeVaultLib.validateWithdrawalCredentials()` has the following checks for `beaconStateRootProof.timestamp`:

```
if (
    beaconStateRootProof.timestamp < node.lastSnapshotTimestamp
        || beaconStateRootProof.timestamp < node.currentSnapshotTimestamp
) revert BeaconTimestampTooOld();
```

As seen from above, only restriction on `beaconStateRootProof.timestamp` is that it cannot be older than the last/ongoing snapshot. This makes it possible for `beaconStateRootProof.timestamp` to be `block.timestamp`.

Later on in the function, the newly added validator's `lastBalanceUpdateTimestamp` is set to `beaconStateRootProof.timestamp` in `NativeVaultLib.validateWithdrawalCredentials()`:

```
validatorDetails.lastBalanceUpdateTimestamp = updateTimestamp;
```

However, if `startSnapshot()` or `validateExpiredSnapshot()` is called after `validateWithdrawalCredentials()` in the same block, the newly added validator cannot be proven with `validateSnapshotProofs()` due to the following check:

```
if (validatorDetails.lastBalanceUpdateTimestamp >= node.currentSnapshotTimestamp) {
    revert ValidatorAlreadyProved();
}
```

This will make it impossible to complete the snapshot as the newly added validator can never be proven, so `snapshot.remainingProofs` will never reach `0`. For example:

- Assume a node owner has no validators.
- In the block where `block.timestamp = 1000`:
  - `validateWithdrawalCredentials()` is called:
    - Assume `beaconStateRootProof.timestamp = block.timestamp`.
    - `validator.lastBalanceUpdateTimestamp = 1000`.
    - `node.activeValidatorCount` is incremented to `1`.
  - `startSnapshot()` is called to start a new snapshot:
    - `snapshot.remainingProofs = 1`
    - `node.currentSnapshotTimestamp = 1000`
- When attempting to prove the validator with `validateSnapshotProofs()`:
  - Both `validatorDetails.lastBalanceUpdateTimestamp` and `node.currentSnapshotTimestamp` are `1000`, so the check shown above reverts.
- As such, the validator can never be proven and `snapshot.remainingProofs` is forever stuck at `1`.

If this occurs, snapshots will be forever DOSed for the node owner.

**Recommendation:** Ensure that `validateWithdrawalCredentials()` cannot be called with `beaconStateRootProof.timestamp` as `block.timestamp` by adding the following check:

```
if (beaconStateRootProof.timestamp == block.timestamp) {
    revert BeaconTimestampIsCurrent();
}
```

Note that even without this check, it is unlikely for `validateWithdrawalCredentials()` to be called with `block.timestamp` as it is difficult to generate proofs for a block root returned by `_getParentBlockRoot()` in a future block.

**Karak:** Fixed in PR 341.

**Renascence:** Verified, the recommended fix was implemented.

### [H-8] Specifying `validatorIndex` as `uint64` allows `BeaconProofsLib.validateValidatorProof()` to pass with incorrect proofs

**Context:**

- BeaconProofsLib.sol#L74-L75
- BeaconProofsLib.sol#L83

**Description:** In `BeaconProofsLib.validateValidatorProof()`, `validatorIndex` is declared as `uint64`:

```
function validateValidatorProof(
    uint64 validatorIndex,
```

However, `validatorIndex` should be `uint40` instead as the maximum length of `validators` in `BeaconState` is `2 ** 40`. Any index greater than `type(uint40).max` is invalid.

This becomes a problem as `validatorIndex` is OR-ed with the other bits in `index`:

```
uint256 index = (CONTAINER_IDX « (VALIDATOR_HEIGHT + 1)) | uint256(validatorIndex);
```

Assuming the rightmost bit in `index` is bit 0, an attacker can set bits 41 to 45 of `validatorIndex` to switch from the `validators` field to certain fields after it in `BeaconState`. You can think of it as modifying `CONTAINER_IDX` to a different value, which would end up proving a different field in `BeaconState`.

For example, assume `CONTAINER_IDX = 15` and `validatorIndex = 0`. `index` would be:

```
(15 « (VALIDATOR_HEIGHT + 1)) | uint256(0) = 0x1e0000000000
```

The same value can be reached with `CONTAINER_IDX = 12` and `validatorIndex = 0x1e0000000000`, since:

```
(12 « (VALIDATOR_HEIGHT + 1)) | uint256(0x1e0000000000) = 0x1e0000000000
```

If `validateValidatorProof()` was called with `validatorIndex = 0x1e0000000000`, the function would end up validating `validatorFields` against the field at index 15 in `BeaconState`, which is `previous_epoch_participation`. This makes it possible for `validateValidatorProof()` to pass with an invalid `validatorFields`.

**Recommendation:** Declare `validatorIndex` as `uint40` instead:

```
  function validateValidatorProof(
-     uint64 validatorIndex,
+     uint40 validatorIndex,
      bytes32[] calldata validatorFields,
```

This change should be reflected throughout the codebase - any variable that represents the validator's index in the beacon chain should be changed to `uint40`:

- BeaconProofsLib.sol#L33-L34
- NativeVaultLib.sol#L20-L22

The unsafe cast from `uint64` to `uint40` at NativeVaultLib.sol#L120 can then be removed.

**Karak:** Fixed in PR 341 and PR 348.

**Renascence:** Verified, the recommended fix was implemented.

### [H-9] Wrong withdraw address verification in `NativeVaultLib.validateWithdrawalCredentials()`

**Context:**

- NativeVaultLib.sol#L162-L167

**Description:** In `NativeVaultLib.validateWithdrawalCredentials()` the withdraw credential verification is the following:

```
if (
    BeaconProofs.getWithdrawalCredentials(validatorFieldsProof.validatorFields)
        != bytes32(abi.encodePacked(bytes1(uint8(1)), bytes11(0), address(this)))
) {
    revert WithdrawalCredentialsMismatchWithNode();
}
```

First two parameters supplied to `abi.encodePacked()` are the prefix `0x01` and 11 zeros bytes as per the withdrawal credential spec, however, the last parameter is the withdrawal address which should be the Native Node, not the Native Vault. **Recommendation:**

```
@@ -161,7 +164,7 @@ library NativeVaultLib {
         // Construct beacon chain withdraw address with current node's payable
         address
         if (
             BeaconProofs.getWithdrawalCredentials(validatorFieldsProof.validatorFiel⌋
             ds)
-                != bytes32(abi.encodePacked(bytes1(uint8(1)), bytes11(0),
address(this)))
+                != bytes32(abi.encodePacked(bytes1(uint8(1)), bytes11(0),
self.ownerToNode[nodeOwner].nodeAddress))
```

**Karak:** Fixed in PR 324.

**Renascence:** Verified, the recommended fix was implemented.

### [H-10] Wrong use of the beacon block root instead of the beacon state root in `NativeVault.validateWithdrawalCredentials()`.

**Context:**

- NativeVault.sol#L186

**Description:** `NativeVaultLib.validateWithdrawalCredentials()` expects the parameter `bytes32 beaconStateRoot`. In `NativeVault.validateWithdrawalCredentials()`, `beaconStateRootProof.beaconStateRoot` is verified, however, It is the beacon block root that is passed to `NativeVaultLib.validateWithdrawalCredentials()`, which is incorrect. The beacon state root should be supplied instead.

**Recommendation:**

```
@@ -183,7 +186,7 @@ contract NativeVault is ERC4626, IBeacon, Pauser, INativeVault,
OwnableRoles, Re
             totalRestakedWei += self.validateWithdrawalCredentials(
                 nodeOwner,
                 beaconStateRootProof.timestamp,
-                _getParentBlockRoot(beaconStateRootProof.timestamp),
+                beaconStateRootProof.beaconStateRoot,
```

**Karak:** Fixed in PR 336.

**Renascence:** Verified, the recommended fix was implemented.

## Medium Risk

### [M-1] `NativeVault.finishWithdrawal()` doesnt reset `withdrawalMap[withdrawalKey]` after executing the withdrawal

**Context:** NativeVault.sol#L261-L262

**Description:** In `NativeVault.finishWithdrawal()`, the withdrawal to execute is fetched with `withdrawalMap[withdrawalKey]`:

```
NativeVaultLib.Storage storage self = _state();
NativeVaultLib.QueuedWithdrawal memory startedWithdrawal =
self.withdrawalMap[withdrawalKey];
```

However, after the pending withdrawal is executed, `self.withdrawalMap[withdrawalKey]` isn't reset in storage. This allows a user to call `finishWithdrawal()` repeatedly with the same `withdrawalKey` to withdraw all his assets, effectively bypassing `MIN_WITHDRAWAL_DELAY`.

**Recommendation:** Consider clearing `withdrawalMap[withdrawalKey]` as such:

```
  NativeVaultLib.Storage storage self = _state();
  NativeVaultLib.QueuedWithdrawal memory startedWithdrawal =
  self.withdrawalMap[withdrawalKey];
+ delete self.withdrawalMap[withdrawalKey];
```

**Karak:** Fixed in PR 327.

**Renascence:** Verified, the recommended fix was implemented.

### [M-2] `Native.startWithdrawal()` can be called repeatedly to queue an infinite number of withdrawals

**Context:** NativeVault.sol#L238

**Description:** The maximum amount of ETH a node owner can withdraw through `NativeVault.startWithdrawal()` is limited by `withdrawableWei()`:

```
if (weiAmount > withdrawableWei(msg.sender)) revert WithdrawMoreThanMax();
```

`withdrawableWei(msg.sender)` is the minimum between the amount of ETH in the caller's native node and the asset equivalent of his shares, so it doesn't exclude the amount of ETH that are currently in pending withdrawals.

As such, users can queue an infinite amount of ETH for withdrawals by repeatedly calling `startWithdrawal()` with `weiAmount = withdrawableWei(msg.sender)`. This allows them to bypass `MIN_WITHDRAWAL_DELAY` for withdrawals in the future as they have an infinite number of pending withdrawals, and can call `finishWithdrawal()` anytime to instantly perform a withdrawal.

**Recommendation:** Consider tracking the amount of ETH in pending withdrawals and subtracting it from `withdrawableWei()` in `startWithdrawal()`.

In `NativeVaultLib`, add a new mapping in `Storage` named `nodeOwnerToWithdrawAmount`, which represents the total amount of assets in pending withdrawals for each node owner:

```
  // mapping of node owner to their withdraw nonce
  mapping(address nodeOwner => uint256 withdrawNonce) nodeOwnerToWithdrawNonce;
+ // mapping of node owner to their total pending withdrawal amount
+ mapping(address nodeOwner => uint256 withdrawAmount) nodeOwnerToWithdrawAmount;
  // mapping of owners' withdraw nonce to pending withdrawals
  mapping(bytes32 ownerWithdrawNonce => QueuedWithdrawal withdrawal) withdrawalMap;
```

In `startWithdrawal()`, subtract `nodeOwnerToWithdrawAmount` from `withdrawableWei()`. Additionally, `nodeOwnerToWithdrawAmount` should be increased by `weiAmount` whenever a new withdrawal is started:

```
- if (weiAmount > withdrawableWei(msg.sender)) revert WithdrawMoreThanMax();

  NativeVaultLib.Storage storage self = _state();
+ if (weiAmount > withdrawableWei(msg.sender) -
self.nodeOwnerToWithdrawAmount[msg.sender]) {
+     revert WithdrawMoreThanMax();
+ }
+ self.nodeOwnerToWithdrawAmount[msg.sender] += weiAmount;
```

In `finishWithdrawal()`, whenever a withdrawal is finished, subtract the amount of assets withdrawn from `nodeOwnerToWithdrawAmount`:

```
if (startedWithdrawal.start == 0) revert WithdrawalNotFound();
if (startedWithdrawal.start + Constants.MIN_WITHDRAWAL_DELAY > block.timestamp) {
    revert MinWithdrawDelayNotPassed();
}

+ self.nodeOwnerToWithdrawAmount[startedWithdrawal.nodeOwner] -=
startedWithdrawal.assets;
```

**Karak:** Fixed in PR 342.

**Renascence:** Verified, the recommended fix was implemented.

### [M-3] `assets > withdrawableWei()` check in `_decreaseBalance()` causes `NativeVault.finishWithdrawal()` to revert when slashing occurs

**Context:**

- NativeVault.sol#L269
- NativeVault.sol#L473

**Description:** When finishing a withdrawal, `NativeVault.finishWithdrawal()` calls `_decreaseBalance()` to decrease the node owner's asset balance:

```
_decreaseBalance(startedWithdrawal.nodeOwner, startedWithdrawal.assets);
```

`_decreaseBalance()` checks that `startedWithdrawal.assets` is not greater than `withdrawableWei()`:

```
if (assets > withdrawableWei(_of)) revert WithdrawMoreThanMax();
```

Note that `withdrawableWei()` returns the minimum between the node owner's native node balance and the assets equivalent of his shares.

However, if Karak operator or the beacon chain slashes the node owner's ETH balance before a withdrawal is finished, it might become impossible for the withdrawal to be executed using `finishWithdrawal()` due to this check.

For example:

- Assume that:
  - A node owner holds `32e18` shares that corresponds to 32 ETH.
  - He is the only node owner in the entire protocol, so `totalSupply` and `totalAssets` are both `32e18` as well.
- Node owner calls `startWithdrawal()` with `weiAmount = 32e18` to withdraw his entire balance.
- Karak operator calls `slashAssets()` to slash 1 ETH, so `totalAssets = 31e18`.
- Node owner calls `finishWithdrawal()` to finish the withdrawal. In `_decreaseBalance()`:
  - `withdrawableWei()` returns 31 ETH.
  - `startedWithdrawal.assets = 32e18` is greater than `withdrawableWei()`, so the check reverts.

If a withdrawal can never be completed using `finishWithdrawal()`, as demonstrated above, the node owner will have to go through the full `MIN_WITHDRAWAL_DELAY` period again to withdraw his assets.

**Recommendation:** Consider removing the `assets > withdrawableWei(_of)` check from `_decreaseBalance()`:

```
  function _decreaseBalance(address _of, uint256 assets) internal {
      NativeVaultLib.Storage storage self = _state();
-     if (assets > withdrawableWei(_of)) revert WithdrawMoreThanMax();
```

In `finishWithdrawal()`, consider limiting the amount of assets withdrawn to `withdrawableWei()` instead of reverting:

```
+ uint256 withdrawableAssets = withdrawableWei(startedWithdrawal.nodeOwner);
+ if (startedWithdrawal.assets > withdrawableAssets) {
+     startedWithdrawal.assets = withdrawableAssets;
+ }
  _decreaseBalance(startedWithdrawal.nodeOwner, startedWithdrawal.assets);
  INativeNode(self.ownerToNode[startedWithdrawal.nodeOwner].nodeAddress).withdraw(
      startedWithdrawal.to, startedWithdrawal.assets
  );
```

**Karak:** Fixed in PR 342.

**Renascence:** Verified, the recommended fix was implemented.

### [M-4] Missing `validatorProof.length` check in `BeaconProofsLib.validateValidatorProof()`

**Context:** BeaconProofsLib.sol#L74-L88

**Description:** In `BeaconProofsLib.validateValidatorProof()`, there no is check on the length of `validatorProof`, which allows an attacker to freely specify the number of proof hashes to be used in `verifyInclusionSha256()`.

If it is shorter than it should be, the number of times `validatorRoot` is hashed will be less. This could potentially cause `validateValidatorProof()` to pass with an invalid `validatorRoot`.

**Recommendation:** The length of `validatorProof` should be the height of the merkleized `Validator` list + the height of the merkleized `BeaconState` container. Consider adding the following check:

```
- if (!Merkle.verifyInclusionSha256(validatorProof, beaconStateRoot, validatorRoot,
index)) {
+ if (
+   validatorProof.length != 32 * ((VALIDATOR_HEIGHT + 1) + BEACON_STATE_HEIGHT) ||
+   !Merkle.verifyInclusionSha256(validatorProof, beaconStateRoot, validatorRoot,
index)
+ ) {
      revert InvalidValidatorFieldsProof();
  }
```

**Karak:** Fixed in PR 342.

**Renascence:** Verified, the recommended fix was implemented.

### [M-5] Function `_getParentBlockRoot` limits the beacon roots lookback window

**Context:**

- NativeVault.sol#L384

**Description:** Since timestamps are 12 seconds apart, the check on line L384 should be `block.timestamp - timestamp >= Constants.BEACON_ROOTS_RING_BUFFER * 12`. Currently, this would limit the `_getParentBlockRoot()` to return only 683 of the latest stored beacon block roots, while the beacon roots contract accommodates 8191.

**Recommendation:** Change the code on line 384 to `Constants.BEACON_ROOTS_RING_BUFFER * 12`. The check could also be entirely removed since the beacon roots contract will also revert on a query that is more than 8191 roots old.

```
# Pseudo code of the beacon roots contract
def get():
    if len(evm.calldata) != 32:
        evm.revert()

    if to_uint256_be(evm.calldata) == 0:
        evm.revert()

    timestamp_idx = to_uint256_be(evm.calldata) % HISTORY_BUFFER_LENGTH
    timestamp = storage.get(timestamp_idx)

    if timestamp != evm.calldata:
        evm.revert()

    root_idx = timestamp_idx + HISTORY_BUFFER_LENGTH
    root = storage.get(root_idx)

    evm.return(root)
```

**Karak:** Fixed in PR 342.

**Renascence:** Verified, the check was removed.

### [M-6] `NativeVaultLib.validateWithdrawalCredentials()` should return the actual balance of the validator

**Context:**

- NativeVaultLib.sol#L169

**Description:** The effective balance of a validator is capped at 32 ETH. If a validator has more than 32 ETH, such as 64 ETH, `getEffectiveBalanceWei()` will return 32 ETH while its balance in `BeaconProofsLib.validateBalance()` would be 64 ETH.

As such, if `NativeVault.validateWithdrawalCredentials()` is called to register a validator that holds more than 32 ETH, only 32 ETH worth of shares will be minted to the `nodeOwner` and added to `totalRestakedETH`. The remaining shares will only be minted in the next snapshot.

This causes the number of shares held by the `nodeOwner` to be temporarily lower than their actual ETH balance until the next snapshot. Additionally, the shares that have not been minted cannot be slashed.

**Recommendation:** Consider specifying `restakedBalanceWei` as the validator's actual balance here, using `validateBalance()`.

**Karak:** Acknowledged. The excess ETH balance isn't restaked so node owners won't be getting rewards for them, so it's fine if it can't be slashed.

**Renascence:** Acknowledged.

### [M-7] Inability to update node implementation in `NativeVault`

**Context:** NativeVault.sol#L85

**Description:** `NativeVault` acts as a beacon for any node deployed through. The function `NativeVault#changeNodeImplementation()` that is used to update the beacon proxy implementation is restricted to the contract owner, which is the `Core` contract. However, the `Core` contract does not include any functionality to invoke `changeNodeImplementation()`. As a result, once a `NativeVault` is deployed, there's no way to update the `nodeImpl` for nodes that rely on `NativeVault` as a beacon.

**Recommendation:** Depending on the intended behavior, make sure the `NativeNode#changeNodeImplementation()` can be called by the respective role within the project, e.g. `MANAGER_ROLE`.

**Karak:** Fixed in PR 322.

**Renascence:** Verified, the issue was fixed by allowing the `MANAGER_ROLE` to call `changeNodeImplementation()`.

## Low Risk

### [L-1] `NativeVault.slashAssets()` could incorrectly return `0`

**Context:** NativeVault.sol#L293-L298

**Description:** In `NativeVault.slashAssets()`, if the amount of assets to slash is greater than the total amount of assets, `totalAssets` is set to `0`:

```
// avoid negative totalAssets if slashing amount is greater than totalAssets
if (totalAssetsToSlash > self.totalAssets) {
    emit Slashed(self.totalAssets);
    self.totalAssets = 0;
    return self.totalAssets;
}
```

However, since `self.totalAssets` is set to `0` before the return statement, it will always return `0` in the block shown above.

**Recommendation:** Consider modifying the logic as such:

```diff
  // avoid negative totalAssets if slashing amount is greater than totalAssets
  if (totalAssetsToSlash > self.totalAssets) {
-     emit Slashed(self.totalAssets);
-     self.totalAssets = 0;
-     return self.totalAssets;
+     totalAssetsToSlash = self.totalAssets;
  }
```

**Karak:** Fixed in PR 338.

**Renascence:** Verified, the recommended fix was implemented.

### [L-2] A node owners `totalRestakedETH` and shares will lag behind his actual restaked balance

**Context:** NativeVault.sol#L422-L425

**Description:** Due to the following check in `_startSnapshot()`, `NativeVault.startSnapshot()` can only be called by a node owner when his native node's ETH balance has increased since the last snapshot:

```
// Calculate unattributed node balance
uint256 nodeBalanceWei = node.nodeAddress.balance - node.creditedNodeETH;

if (throwIfNoBalanceChange && nodeBalanceWei == 0) revert NoBalanceUpdateToSnapshot();
```

However, a node owner might want to start a snapshot even if his native node's balance hasn't changed since the last snapshot. For example, if his validator's balance on the beacon chain increased, starting a snapshot would increase `totalRestakedETH` and mint more shares to him to reflect this change.

`startSnapshot()` cannot be called by the node owner until either:

1. 7 days has passed, causing his last snapshot to expire.
2. A partial withdrawal is executed on the beacon chain to withdraw his validator's excess balance to the native node.

Note that the duration of (2) depends entirely on the state of the beacon chain, and can take up to multiple days to occur.

Therefore, since the node owner has to wait for a period of time before `startSnapshot()` can be called, his `totalRestakedETH` value and number of shares will temporarily lag behind the actual amount of ETH he has restaked.

**Recommendation:** Consider removing this check from `_startSnapshot()`:

```
- if (throwIfNoBalanceChange && nodeBalanceWei == 0) revert
NoBalanceUpdateToSnapshot();
```

```
- function _startSnapshot(NativeVaultLib.NativeNode storage node, bool
throwIfNoBalanceChange, address nodeOwner)
+ function _startSnapshot(NativeVaultLib.NativeNode storage node, address nodeOwner)
      internal
  {
```

**Karak:** Fixed in PR 338. Instead of deciding on our own, we can take this from the user itself. So, if they don't want to go ahead with a snapshot with 0 node balance update, they can pass in the option.

**Renascence:** Verified, the issue was fixed by allowing the user to specify `throwIfNoBalanceChange` when calling `NativeVault.startSnapshot()`.

### [L-3] `NativeVault.startWithdrawal()` should ensure the node owners last snapshot has not expired

**Context:** NativeVault.sol#L237

**Description:** In the current implementation of the code, `NativeVault.startWithdrawal()` can be called by the node owner to start a withdrawal regardless of when his last snapshot was taken.

This seems to be known by the team in the following TODO:

```
// TODO: make recent snapshot compulsory
```

However, at the very least, `startWithdrawal()` should ensure that the node owner's last snapshot isn't expired. In the event where slashing occurs and the node owner wishes to withdraw all his funds, he can just call `startWithdrawal()` to withdraw his entire balance without updating his snapshot.

This forces the protocol to call `validateExpiredSnapshot()` on his behalf to transfer the remaining funds left behind from slashing to the `slashStore`.

**Recommendation:** Add the following check to `startWithdrawal()`:

```
  NativeVaultLib.Storage storage self = _state();
+ NativeVaultLib.NativeNode storage node = self.ownerToNode[msg.sender];
+
+ if (block.timestamp >= node.lastSnapshotTimestamp + Constants.SNAPSHOT_EXPIRY) {
+     revert SnapshotNotExpired();
+ }
```

**Karak:** Fixed in PR 338.

**Renascence:** Verified, the recommended fix was implemented.

### [L-4] The `operator` of `NativeVault` is not stored in `NativeVaultLib.Storage`

**Context:**

- NativeVault

**Description:** In `NativeVault.initialize()`, the `operator` is stored only in `VaultLib.Config`, `self.operator` is not assigned and remains `address(0)` inside `NativeVaultLib.Storage`. The parts of the code that become affected are:

1. `NativeVaultLib.deployNode()`: The calculation of the `salt` for the same `msg.sender` will be the same across Native Vaults even if the operator is different. This does not seem to lead to anything major.

2. `NativeVault.startWithdrawal()` and `NativeVault.finishWithdrawal()` functions will emit events with `address(0)` as the operator.

**Recommendation:** Assign `self.operator` to the operator in `NativeVault.initialize()`, the name and symbol variables of the Vault in `NativeVaultLib.Storage` can be assigned as well.

```
+        self.name = _name;
+        self.symbol = _symbol;
+        self.operator = _operator;
```

**Karak:** Fixed in PR 338.

**Renascence:** Verified, the issue was fixed by removing `name`, `symbol` and `operator` from `NativeVaultLib.Storage` and can be accessing them from `VaultLib.Config` through the `_config()` function.

### [L-5] Redundant pausing modifier in `NativeVault.createNode()`

**Context:** NativeVault.sol#L97-L99

**Description:** The `createNode()` function uses both `whenNotPaused()` and `whenFunctionNotPaused(Constants.PAUSE_NATIVEVAULT_CREATE_NODE)`, leading to redundancy. If `createNode()` should be paused when any function in the protocol is paused, remove `whenFunctionNotPaused`, as `whenNotPaused` covers this. If it should only be paused for `PAUSE_NATIVEVAULT_CREATE_NODE`, remove `whenNotPaused`.

**Recommendation:** Depending on the intended behavior, remove either the `whenNotPaused` or `whenFunctionNotPaused` modifier from the `createNode()` function.

**Karak:** Fixed in PR 338.

**Renascence:** Verified, the `whenNotPaused` modifier was removed.

## Informational

### [I-1] `NativeVault.validateWithdrawalCredentials()` should call `_increaseBalance()` to avoid an unsafe `int256` cast

**Context:** NativeVault.sol#L191

**Description:** In `NativeVault.validateWithdrawalCredentials()`, `totalRestakedWei` (which is a `uint256`) is cast directly to `int256` and passed to `_updateBalance()`:

```
_updateBalance(nodeOwner, int256(totalRestakedWei));
```

However, since `totalRestakedWei` can never be negative, `_increaseBalance()` should be used directly instead of `_updateBalance()` to avoid casting to `int256`.

**Recommendation:** Consider calling `_increaseBalance()` instead:

```
  _updateBalance(nodeOwner, int256(totalRestakedWei));
+ _increaseBalance(nodeOwner, totalRestakedWei);
```

**Karak:** Fixed in PR 337.

**Renascence:** Verified, the recommended fix was implemented.

### [I-2] `STATE_ROOT_IDX` and `BEACON_STATE_ROOT_IDX` can be combined in a single constant

**Context:**

- BeaconProofsLib.sol#L9
- BeaconProofsLib.sol#L11

**Description:** `STATE_ROOT_IDX` and `BEACON_STATE_ROOT_IDX` both represent the same value, which is the index of `state_root` in `BeaconBlockHeader`.

**Recommendation:** Consider removing `STATE_ROOT_IDX` and using `BEACON_STATE_ROOT_IDX` in `BeaconProofsLib.validateBalanceContainer()`.

**Karak:** Fixed in PR 337.

**Renascence:** Verified, the recommended fix was implemented.

### [I-3] Unused `NativeNode` pause functions by `NativeVault`

**Context:**

- NativeNode.sol#L37-L45

**Description:** The functions `NativeNode.pause()` and `NativeNode.unpause()` are never called by the `owner` of the contract, which is `NativeVault`.

**Recommendation:** Consider including the following functions in `NativeVault` if pausing and unpausing of `NativeNode` is intended behavior.

```
function pauseNode(INativeNode node, uint256 map) external
onlyRolesOrOwner(Constants.MANAGER_ROLE) {
    node.pause(map);
}

function unpauseNode(INativeNode node, uint256 map) external
onlyRolesOrOwner(Constants.MANAGER_ROLE) {
    node.unpause(map);
}
```

**Karak:** Acknowledged. We don't need `NativeNode` to be pausable since the only function it has is `withdraw()`, and that too can be only called by the owner which will always be the `NativeVault`. If the withdraw function in `NativeVault` is paused, then it implies that the native node's withdraw is paused.

**Renascence:** It's fine to not have `NativeNode` be pausable, but the `MANAGER_ROLE` would have to pause `finishWithdrawal()`, alongside `startSnapshot()` and `validateExpiredSnapshot()` as well since `_startSnapshot()` also withdraws from the native node for slashing.

### [I-4] Reverting `receive()` function in `NativeNode` is redundant

**Context:**

- NativeNode.sol#L48-L50

**Description:** If `NativeNode` doesn't have a `receive()` function, attempts to transfer ETH, that are not coming from the BeaconChain, will fail. The inclusion of a reverting `receive()` function is redundant.

**Recommendation:**

```
- /// @notice Direct deposit to NativeNode address is not allowed to limit sources of
unattributed ETH
- receive() external payable {
-     revert DirectDepositToNode();
- }
```

**Karak:** Fixed in PR 337.

**Renascence:** Verified, the recommended fix was implemented.

### [I-5] Missing reentrancy guards for functions in `NativeVault`, `NativeNode` and `SlashStore`

**Context:**

- NativeVault.sol#L113
- NativeVault.sol#L127
- NativeVault.sol#L164
- NativeVault.sol#L200
- NativeNode.sol#L56
- SlashStore.sol#L32

**Description:** The functions `NativeVault.startSnapshot()`, `NativeVault.validateSnapshotProofs()`, `NativeVault.validateWithdrawalCredentials()`, `NativeVault.validateExpiredSnapshot()`, `NativeNode.withdraw()`, and `SlashStore.withdraw()` could benefit from reentrancy guards as protective measures.

**Recommendation:** Add `nonReentrant` modifiers to the listed functions.

**Karak:** Fixed in PR 337 and PR 346.

**Renascence:** Verified, the `nonReentant` modifier was added to the functions listed above.

### [I-6] Code improvements

**Description / Recommendation**

1. Pauser.sol#L76

The line of code can be simplified to:

```diff
- if (((~self._paused) & (~map)) != (~self._paused)) revert
AttemptedPauseWhileUnpausing();
+ if (self._paused & map != map) revert AttemptedPauseWhileUnpausing();
```

2. Pauser.sol#L36-L42

Consider using the names `__Pauser_init()` and `__Pauser_init_unchained()` to be consistent with the upgradeable contracts' init function name pattern and avoid confusion in child contracts that use it. The init functions in upgradeable contracts normally follow the pattern `__{ContractName}_init` and `__{ContractName}_init_unchained`.

**Karak:** Fixed in PR 337.

**Renascence:** Verified, the recommended fix was implemented.

## 4.1 Centralization Risks

### 4.1.1 BENEFACTOR must be trusted

Currently, the `BENEFACTOR` is responsible for adding allocations as merkle root in the `MerkleVester` contract. However, any other party could be supplying the funds through `MerkleVester.fund()`. The `BENEFACTOR` can add a rogue `merkleRoot` and withdraw funds from the contract. Note that under normal operation, the `BENEFACTOR` is trusted to add valid allocations that do not break the assumptions code in the contract (ie. each allocation only takes up a 100% of its own total allocation) - the contract has no way of verifying that the allocation in the merkle root is correct. Any incorrectly added allocation could result in unexpected behavior.

The `BENEFACTOR` can withdraw funds from the contract if the `defundFeature` is `true` through `MerkleVester.defund()`. If the `defundFeature` is `false`, the `BENEFACTOR` can still withdraw funds as long as a call to `_checkOrSetDistributionState()` hasn't recorded the new obligations in `totalKnownObligations`. In cases where an allocation is cancelable or revocable, the `BENEFACTOR` can use `MerkleVester.cancel()` or `MerkleVester.revoke()` and then use `MerkleVester.defund()` to acquire the funds.