Karak Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, rvierdiiev, Shaka

June 30th 2024 - July 6th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **karak-network/karak-restaking** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. 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 on 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.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 8aad9ad5592edec06528c5fc75a8eaee395850d2

fixes review commit hash - 0d59567becac7887b79ee70dcda0063a458ad791

Scope

The following smart contracts were in scope of the audit:

- `NativeNode`
- `NativeVault`
- `SlashStore`
- `NativeVaultLib`
- `BeaconProofsLib`
- `MerkleProofs`
- `Constants`
- `Errors`
- `Events`

7. Executive Summary

Over the course of the security review, 0xunforgiven, rvierdiiev, Shaka engaged with Karak to review Karak. In this period of time a total of **13** issues were uncovered.

Protocol Summary

Protocol Name	Karak
Repository	https://github.com/karak-network/karak-restaking
Date	June 30th 2024 - July 6th 2024
Protocol Type	Restaking protocol

Findings Count

Severity	**Amount**
High	7
Medium	4
Low	2
Total Findings	**13**

Summary of Findings

ID	Title	Severity	Status
[H-01]	User can prevent balance updates by withdrawing	High	Resolved
[H-02]	Broken balance update if a slash event happens	High	Resolved
[H-03]	activeValidatorCount is never set or increased	High	Resolved
[H-04]	_increaseBalance() mints fewer shares than expected	High	Resolved
[H-05]	Transferring NativeVault tokens can break some functionalities	High	Resolved
[H-06]	finishWithdrawal() doesn't remove withdrawal info	High	Resolved
[H-07]	Code should use state root to validate withdraw credentials	High	Resolved
[M-01]	The User can deny paying slashed tokens	Medium	Resolved
[M-02]	Operator's manager can pause vault actions	Medium	Resolved
[M-03]	Wrong withdrawal address check	Medium	Resolved
[M-04]	Preventing balance updates by adding a new validator in the current block	Medium	Acknowledged
[L-01]	Transferring NativeVault tokens to non-node owners turns them into useless assets	Low	Resolved

[L-02]	Node owners can start withdrawal and execute it just in case of slashing	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] User can prevent balance updates by withdrawing

Severity

Impact: Medium

Likelihood: High

Description

When `finishWithdraw()` is called the ETH amount is transferred from the native node contract's balance but the code doesn't decrease the value of the `creditedNodeETH` so its value would be higher than it should be and `_startSnapshot()` would revert because of underflow:

```
// Calculate unattributed node balance

                uint256 nodeBalanceWei = node.nodeAddress.balance - node.creditedNode
```

The issue would happen to all users who withdraw tokens and malicious users can do it intentionally to prevent balance updates (when they are slashed in the beacon chain)

Recommendations

Update the value of the `creditedNodeETH` in `finishWithdraw()`.

[H-02] Broken balance update if a slash event happens

Severity

Impact: Medium

Likelihood: High

Description

When `_transferToSlashStore()` is called the slash ETH amount is transferred from the native node contract's balance but the code doesn't decrease the value of the `creditedNodeETH` so its value would be higher than it should be and `_startSnapshot()` would revert because of underflow:

```
// Calculate unattributed node balance

                uint256 nodeBalanceWei = node.nodeAddress.balance - node.creditedNode
```

Recommendations

Update the value of the `creditedNodeETH` in `_transferToSlashStore()`.

[H-03] `activeValidatorCount` is never set or increased

Severity

Impact: Medium

Likelihood: High

Description

In the `NativeVault` contract `Storage.ownerToNode[nodeOwner].activeValidatorCount` keeps track of the total validators with withdrawal credentials pointed to a specific node.

This value is decreased in `NativeVaultLib.validateSnapshotProof` when the balance of a validator in the Beacon Chain is zero.

```
File: NativeVaultLib.sol

        if (newBalanceWei == 0) {
@>          self.ownerToNode[nodeOwner].activeValidatorCount--;
            validatorDetails.status = ValidatorStatus.WITHDRAWN;

            emit ValidatorWithdrawn
              (nodeOwner, nodeAddress, timestamp, validatorIndex);
        }
```

The value is used in `_startSnapshot` to set the `remainingProofs` field of the `Snapshot` struct.

```
File: NativeVault.sol

        NativeVaultLib.Snapshot memory snapshot = NativeVaultLib.Snapshot({
            parentBeaconBlockRoot: _getParentBlockRoot(uint64(block.timestamp)),
            nodeBalanceWei: nodeBalanceWei,
            balanceDeltaWei: 0,
@>          remainingProofs: node.activeValidatorCount
        });
```

However, `activeValidatorCount` is never set or increased. As a result `remainingProofs` will always be initialized to zero, causing the following outcomes:

- `validateSnapshotProofs` will always revert due to underflow error.

```
File: NativeVault.sol

        for (uint256 i = 0; i < balanceProofs.length; i++) {
(...)
@>          snapshot.remainingProofs--;
            snapshot.balanceDeltaWei += balanceDeltaWei;
        }
```

- `_updateSnapshot` will always evaluate to `true` the condition `snapshot.remainingProofs == 0`. As a result `node.creditedNodeETH` can be updated twice: once on `startSnapshot` and another called `validateSnapshotProofs` with an empty `balanceProofs` array.

```
File: NativeVault.sol

@>      if (snapshot.remainingProofs == 0) {
            int256 totalDeltaWei = int256
              (snapshot.nodeBalanceWei) + snapshot.balanceDeltaWei;

@>          node.creditedNodeETH += snapshot.nodeBalanceWei;

            node.lastSnapshotTimestamp = node.currentSnapshotTimestamp;
            delete node.currentSnapshotTimestamp;
            delete node.currentSnapshot;

            _updateBalance(nodeOwner, totalDeltaWei);
            emit SnapshotFinished(
              nodeOwner,
              node.nodeAddress,
              node.lastSnapshotTimestamp,
              totalDeltaWei
            );
        } else {
            node.currentSnapshot = snapshot;
        }
```

Recommendations

Add the following line in `NativeVaultLib.validateWithdrawalCredentials`:

```
validatorDetails.status = NativeVaultLib.ValidatorStatus.ACTIVE;

                validatorDetails.validatorIndex = validatorFieldsProof.validatorProof
        validatorDetails.lastBalanceUpdateTimestamp = updateTimestamp;
        validatorDetails.restakedBalanceWei = restakedBalanceWei;

                self.ownerToNode[nodeOwner].validatorPubkeyHashToDetails[validatorPub
+       self.ownerToNode[nodeOwner].activeValidatorCount++;
```

[H-04] `_increaseBalance()` mints fewer shares than expected

Severity

Impact: High

Likelihood: Medium

Description

To increase the balance of a node owner, the `NativeVault._increaseBalance` function is called. This function increases the total assets, calculates the shares

to be minted, mints the shares for the node owner, and updates the `totalRestakedETH` of the node owner.

However, the order of the operations is not correct, as updating the `totalAssets` before the calculation of the shares to be minted causes the number of shares received to be lower than it should be.

```
File: NativeVault.sol

    function _increaseBalance(address _of, uint256 assets) internal {
        NativeVaultLib.Storage storage self = _state();
        if (assets + self.totalAssets > maxDeposit
          (_of)) revert DepositMoreThanMax();
 @>     self.totalAssets += assets;
        uint256 shares = convertToShares(assets);
        _mint(_of, shares);
        self.ownerToNode[_of].totalRestakedETH += assets;
        emit IncreasedBalance(self.ownerToNode[_of].totalRestakedETH);
    }
```

Proof of concept

The state of the vault is as follows:

- totalAssets: 32 ETH
- totalSupply: 32e18
- exchange rate: 1:1

Alice's balance is increased by 32 ETH, so she should receive 32e18 shares. But this is not the case. The calculation is as follows:

- totalAssets = 32 ETH + 32 ETH = 64 ETH
- shares = assets * totalSupply / totalAssets = 32 ETH * 32e18 / 64 ETH = 16e18
- totalSupply = 32e18 + 16e18 = 48e18

Alice receives 16e18 shares. Converting them to assets we have:

- assets = shares * totalAssets / totalSupply = 16e18 * 64 ETH / 48e18 = 21.33 ETH

Alice has lost 10.67 ETH.

Recommendations

```
function _increaseBalance(address _of, uint256 assets) internal {
        NativeVaultLib.Storage storage self = _state();
        if (assets + self.totalAssets > maxDeposit
          (_of)) revert DepositMoreThanMax();
-       self.totalAssets += assets;
        uint256 shares = convertToShares(assets);
        _mint(_of, shares);
+       self.totalAssets += assets;
        self.ownerToNode[_of].totalRestakedETH += assets;
        emit IncreasedBalance(self.ownerToNode[_of].totalRestakedETH);
    }
```

[H-05] Transferring `NativeVault` tokens can break some functionalities

Severity

Impact: High

Likelihood: Medium

Description

`NativeVault` tokens (shares) can be transferred from the node owner to another address. This can cause different issues in the behavior of the protocol. Here there are some examples:

1. The sender of the tokens might be slashed with more assets than he should be, as his balance would have decreased, but not the `node.totalRestakedETH` value.

```
File: NativeVault.sol

    function _transferToSlashStore(address nodeOwner) internal {
        NativeVaultLib.Storage storage self = _state();
        NativeVaultLib.NativeNode storage node = self.ownerToNode[nodeOwner];

        // slashed ETH = total restaked ETH
        //(node + beacon) - share price equivalent ETH
  @>    uint256 slashedAssets = node.totalRestakedETH - convertToAssets
    (balanceOf(nodeOwner));
```

2. If the recipient of the tokens is not a node owner who has enough balance in their node, he will not be able to use the shares to withdraw or do anything else, turning the shares into a useless asset.

3. If the recipient is a node owner and there has been a slashing event in the protocol, the calculation of `slashedAssets` can underflow, as his balance would have increased, but not the `node.totalRestakedETH` value. This could potentially be used by a node owner to prevent `validateExpiredSnapshot` from being executed in the case of a slashing event in one of his validators in the Beacon Chain.

Recommendations

Override the `transfer` and `transferFrom` functions in order to disallow transfers of the token.

[H-06] `finishWithdrawal()` doesn't remove withdrawal info

Severity

Impact: High

Likelihood: Medium

Description

In order to withdraw funds, the user needs to make a withdrawal request. This will store information about the amount that the user wants to withdraw and the request will be queued for execution.

After `MIN_WITHDRAWAL_DELAY` passes user has the ability to call `finishWithdrawal` to get funds. The reason why the user needs to wait `MIN_WITHDRAWAL_DELAY` is to make sure the user can't avoid slashing by frontrunning the slash action.

The problem is that `finishWithdrawal` function doesn't delete information about withdrawal after it's executed and as a result user can execute it as many times as he wishes and he doesn't need to wait `MIN_WITHDRAWAL_DELAY` anymore.

Recommendations

Delete withdrawal information, after you execute it.

[H-07] Code should use state root to validate withdraw credentials

Severity

Impact: Medium

Likelihood: High

Description

In the function `validateWithdrawalCredentials()` code validates the state root's proof but then it uses the block's root to verify the validator's information:

```
totalRestakedWei += self.validateWithdrawalCredentials(
                nodeOwner,
                beaconStateRootProof.timestamp,
                _getParentBlockRoot(beaconStateRootProof.timestamp),
                validatorFieldsProofs[i]
            );
```

Because of this issue, no validator can be added to the system.

Recommendations

Validator information is stored inside the state root and the code should use `beaconStateRoot` when calling `validateWithdrawalCredentials()`.

8.2. Medium Findings

[M-01] The User can deny paying slashed tokens

Severity

Impact: High

Likelihood: Low

Description

Slash tokens only get transferred when `startSnapshot()` is called by the user or by others when the snapshot is expired. The issue is that if the user has no active validator then it won't be possible to call `validateExpiredSnapshot()` and slash tokens would stuck in the native node. This is the POC:

1. User1 withdraws all his validator's balance into the native node and updates the snapshot.
2. There's a slash event and native vault's users get slashed.
3. Now if User1 won't call `startSnapshot()` and the snapshot expires then others can't call `validateExpiredSnapshot()` and slashed tokens won't get transferred.

The reason why others can't call `validateExpiredSnapshot()` is because of this check:

```
if
          (validatorDetails.status != NativeVaultLib.ValidatorStatus.ACTIVE) revert Va
```

Recommendations

Fix the check in the `validateExpiredSnapshot()` and check the expiration time of the previous snapshot's timestamp.

[M-02] Operator's manager can pause vault actions

Severity

Impact: Medium

Likelihood: Medium

Description

When NativeVault is created, its manager is granted a role that allows him to pause different actions. With such power, the manager can break some functionality of the protocol. An example of such actions is slashing. The manager can pause it and then Core contract will not be able to slash funds from the operator. Another example: the manager can forbid users to withdraw.

Recommendations

Think about removing the manager role, as the operator is not a fully trusted entity.

[M-03] Wrong withdrawal address check

Severity

Impact: Medium

Likelihood: Medium

Description

When a new validator is added to the owner using `validateWithdrawalCredentials` function, then its withdrawal recipient address is checked to be equal to `NativaVault`.

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

This is incorrect because the protocol needs all rewards to be sent to the `NativeNode` of the validator's owner, that's why this address should be checked as a withdrawal recipient.

Recommendations

Check that the withdrawal recipient is NativeNode address of the validator's owner.

[M-04] Preventing balance updates by adding a new validator in the current block

Severity

Impact: Medium

Likelihood: Medium

Description

The balance of the user's validators can be updated by creating a new snapshot and proving the balance of the validators. If a user doesn't start a new snapshot in time then anyone can start a snapshot for that user and update the user's balance. Users have an incentive to not update their balances if they get slashed in the beacon chain. A user can block the snapshot finalization and snapshot creation process and stop balance updates by performing this attack. The issue is that the `validateSnapshotProofs()` code doesn't allow updating the validators balance if the validator balance is updated after the snapshot creation:

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

And a snapshot only finalizes when the `remainingProofs` is 0. So This is the attack POC:

1. User1 who has slashed on the beacon chain wants to prevent balance updates.
2. User1 would create new validator and call `validateWithdrawalCredentials()` and `startSnapshot()` add the same Ethereum block.
3. As a result for the new validator: `validatorDetails.lastBalanceUpdateTimestamp` would equal to `node.currentSnapshotTimestamp`.
4. Because a new validator has been counted on the `remainingProofs` and it's not possible to prove its balance for the current snapshot then `remainingProofs` would never reach 0 and the current snapshot would never finish and the balance update process would be broken.

Recommendations

It's best to change `validateWithdrawalCredentials()` and not allow the creation of the new validators for the current block's timestamp. Changing the above condition to `validatorDetails.lastBalanceUpdateTimestamp > node.currentSnapshotTimestamp` in `validateSnapshotProofs()` will create another issue.

8.3. Low Findings

[L-01] Transferring `NativeVault` tokens to non-node owners turns them into useless assets

`NativeVault` tokens (shares) can be transferred from the node owner to another address. However, if the recipient address is not a node owner that has enough balance in their node, they will not be able to use the shares to withdraw or do anything else, turning the shares into a useless asset.

It is recommended to override the `transfer` and `transferFrom` functions in order to disallow transfers of the token.

[L-02] Node owners can start withdrawal and execute it just in case of slashing

Node owners can queue a withdrawal calling `NativeVault.startWithdrawal` and, after `MIN_WITHDRAW_DELAY` has passed, call `NativeVault.finishWithdrawal` to withdraw the funds. As the documentation states, "This withdrawal queue is meant to prevent a bad actor from indulging in slashable behavior and immediately withdrawing their funds".

However, `startWithdrawal` does not subtract the amount from the user's balance nor imposes a deadline to complete the withdrawal. This means that the node owner will keep accruing rewards and can call `finishWithdrawal` at any time once `MIN_WITHDRAW_DELAY` has passed.

Node owners can call `startWithdrawal` and leave the withdrawal queued indefinitely, so that they can front-run a slashing event, just before it happens, by calling `finishWithdrawal`.

What is more, there is no limit to the number of withdrawals queued, so they can keep calling `startWithdrawal` with the updated amount every time they receive rewards from the Beacon Chain.

We recommend:

- Do not allow node owners to queue a new withdrawal if there is already one in progress.
- Implement a deadline for withdrawals, so that node owners have a limited time to call `finishWithdrawal`, after which the withdrawal is considered expired.

This deadline should offer a withdrawal window that is not too long in comparison to the `MIN_WITHDRAW_DELAY`. The reasoning is the following:

Having `MIN_WITHDRAW_DELAY` of 7 days and a withdrawal deadline of `block.timestamp + 21 days` would mean by calling `startWithdrawal` every 21 days, a node owner will be able to withdraw their funds, and thus front-run a slashing event, 2/3 of the time.