

# Karak

## Smart Contract Security Assessment

Version 1.0

Audit dates: Jun 10 — Jun 17, 2024

Audited by: xiaoming9090  
thereksfour

# Contents

## 1. Introduction

1.1 About Zenith

1.2 Disclaimer

1.3 Risk Classification

## 2. Executive Summary

2.1 About Karak

2.2 Scope

2.3 Audit Timeline

2.4 Issues Found

## 3. Findings Summary

## 4. Findings

4.1 Low Risk

4.2 Informational

# 1. Introduction

## 1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at <https://code4rena.com/zenith>.

## 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

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

# 2. Executive Summary

## 2.1 About Karak

Karak is the universal restaking layer that makes it easy to provide cryptoeconomic security with any asset, and unlocks a new design space for developers to seamlessly and securely create innovative infrastructure designs. Karak enables protocols to tap into robust and secure trust networks from day one, significantly lowering the barrier to securing new protocols and eliminating the need for protocols to incentivize their own validator sets with a

highly dilutive reward mechanism, making the process of bootstrapping security more scalable, accessible, and affordable.

## 2.2 Scope

Repository	<a href="#">Risk-Harbor/karak-restaking/</a>
Commit Hash	<a href="#">16be0e4f3e64797193a00395e57d5a11a8f2f54d</a>

## 2.3 Audit Timeline

DATE	EVENT
Jun 10, 2024	Audit start
Jun 17, 2024	Audit end
Oct 25, 2024	Report published

## 2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	3
Informational	1
Total Issues	4

## 3. Findings Summary

ID	DESCRIPTION	STATUS
L-1	<code>`implementation(address(0))` should not return <code>`self.vaultImpl`</code></code>	Resolved

L-2	Standard implementation can be set to reserved address ( <code>address(1)</code> )	Resolved
L-3	<code>`slashablePercentageWad`</code> can exceed 100%	Resolved
I-1	Ambiguity in the DSS's slashable percentage returned from <code>`Core.getDssSlashablePercentageWad`</code> function	Resolved

## 4. Findings

### 4.1 Low Risk

A total of 3 low risk findings were identified.

[L-1] ``implementation(address(0))`` should not return ``self.vaultImpl``

---

Severity: Low

Status: Resolved

---

Context:

- [Core.sol#L308-L316](#)

Description:

When deploying a Vault, if `implementation` is `address(0)`, `implementation` is replaced with `DEFAULT_VAULT_IMPLEMENTATION_FLAG` and assigned to `vaultToImplMap`, which makes valid `vaultToImplMap` will not be `address(0)`.

```
if (implementation == address(0)) {  
    // Allows us to change all the standard vaults to a new  
    implementation  
    implementation = Constants.DEFAULT_VAULT_IMPLEMENTATION_FLAG;  
}
```

And in `changeImplementationForVault()`, a `vaultToImplMap` of `address(0)` is considered invalid.

```
if (self.vaultToImplMap[vault] == address(0)) revert  
VaultNotAChildVault();
```

However, in `implementation()`, `address(0)` is considered valid and `self.vaultImpl` is returned.

This would make the `implementation()` of any non-Vault address be `self.vaultImpl`, which might have some side effects out of scope.

Recommendation:

It is recommended to make `implementation(address(0))` return `address(0)`

```

function implementation(address vault) public view returns (address)
{
    CoreLib.Storage storage self = _self();
    address vaultImplOverride = self.vaultToImplMap[vault];

-    if (vaultImplOverride ==
Constants.DEFAULT_VAULT_IMPLEMENTATION_FLAG || vaultImplOverride ==
address(0)) {
+    if (vaultImplOverride ==
Constants.DEFAULT_VAULT_IMPLEMENTATION_FLAG) {
        return self.vaultImpl;
    }
    return vaultImplOverride;
}

```

Karak: [PR-287](#)

Zenith: Verified.

## [L-2] Standard implementation can be set to reserved address (`address(1)`)

Severity: Low

Status: Resolved

### Context:

- [Core.sol#L171](#)

### Description:

`address(1)` is a reserved address for default implementation within the protocol. The protocol should not allow anyone to set the standard implementation address to `address(1)`.

```
File: Core.sol
171:     function changeStandardImplementation(address newVaultImpl)
external onlyOwner { // @audit-ok
172:         if (newVaultImpl == address(0)) revert ZeroAddress();
173:         CoreLib.Storage storage self = _self();
174:         self.vaultImpl = newVaultImpl;
175:         emit UpgradedAllVaults(newVaultImpl);
176:     }
```

### Recommendation:

To avoid any unexpected error or mistake, consider adding the following check since `address(1)` is reserved for default implementation.

```
+ if (newVaultImpl == Constants.DEFAULT_VAULT_IMPLEMENTATION_FLAG) revert
ReservedAddress();
```

Karak: Fixed in [PR 287](#)

Zenith: Verified



### [L-3] `slashablePercentageWad` can exceed 100%

Severity: Low

Status: Resolved

#### Context:

- [Core.sol#L266](#)

#### Description:

The `slashablePercentageWad` should never exceed 100%. However, it was found that the DSS can set it to a percentage beyond 100% via the `Core.setDssSlashablePercentageWad` function.

```
File: Core.sol
266:     function setDssSlashablePercentageWad(uint256
slashablePercentageWad) external {
267:         CoreLib.Storage storage self = _self();
268:         uint256 currentSlashablePercentageWad =
self.dssSlashablePercentageWad[IDSS(msg.sender)];
269:         if (currentSlashablePercentageWad != 0) revert
SlashingPercentAlreadySet();
270:         self.dssSlashablePercentageWad[IDSS(msg.sender)] =
slashablePercentageWad;
271:     }
```

#### Recommendation:

To prevent any potential edge cases, ensure the DSS cannot set its slashable percentage to more than 100%.

```
+ require(slashablePercentageWad <= Constants.MAX_SLASHING_PERCENT_WAD)
```

In addition, consider disallowing DSS to set its slashable percentage to zero (if applicable).

Karak: Fixed in [PR-318](#)

Zenith: Verified

## 4.2 Informational

A total of 1 informational findings were identified.

### [I-1] Ambiguity in the DSS's slashable percentage returned from ``Core.getDssSlashablePercentageWad`` function

Severity: Informational

Status: Resolved

#### Context:

- [Core.sol#L354](#)

#### Description:

Within the `SlasherLib.validate` function, if the `dssSlashablePercentageWad` is not initialized (equal to zero), the slashable percentage is 100%.

```
uint256 maxSlashingWad = self.dssSlashablePercentageWad[dss] == 0
    ? Constants.MAX_SLASHING_PERCENT_WAD
    : self.dssSlashablePercentageWad[dss];
```

This might cause some confusion for users who rely on the `getDssSlashablePercentageWad` function to determine a DSS's slashable percentage. When this function returns zero, it is unclear whether the DSS's slashable percentage is 0% or 100%. Users might think that the DSS's slashable percentage is 0%, while, in fact, it is 100%.

#### Recommendation:

Consider having a variable that keeps track of whether the `dssSlashablePercentageWad` has already been initialized so it can be used within the `SlasherLib.validate` function to only return `MAX_SLASHING_PERCENT_WAD` (100%) if it is uninitialized.

Alternatively, document this behavior in the NatSpec of this function.

Karak: Fixed in [PR-287](#)

Zenith: Verified