REN∆SCENCE

Karak Restaking Audit Report

Version 2.0

Audited by:

HollaDieWaldfee

bytes032

alexxander

June 9, 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

In Karak Restaking, the key participants are Stakers, Node Operators, and Distributed Secure Services (DSS). Operators create Vaults in which Stakers lock their funds and trust an Operator in exchange for the Operator's earned rewards. Operators can register with one or multiple Distributed Secure Services (DSS). Operators take on the responsibility to perform tasks for the DSS in exchange for rewards. If an Operator fails to perform the assigned tasks, the DSS can use a slashing mechanism to allocate part of the Operator's funds to a Slashing Handler, which will then compensate the DSS according to the implementation of the specific Slashing Handler.

2.2 Overview

Project	Karak Restaking
Repository	karak-restaking
Commit Hash	361c68347e5b...
Mitigation Hash	5d2103da2e5a...
Date	17 May 2024 - 20 May 2024

2.3 Issues Found

Severity	**Count**
High Risk	9
Medium Risk	5
Low Risk	8
Informational	5
Total Issues	**27**

3 Findings Summary

ID	Description	Status
H-1	Redemptions in Vault can be cancelled by anyone	Resolved
H-2	Percentage based allocations introduce race conditions and DSS can slash more funds than operator has staked to it	Open
H-3	`Core`: DSS is called with wrong hook functions	Resolved
H-4	`Core`: `callHookIfInterfaceImplemented()` is called with wrong interface IDs	Resolved
H-5	`Core`: `callHookIfInterfaceImplemented()` is called with wrong `canFail` parameters	Resolved
H-6	`Vault`: Redemption process can be gamed to avoid slashing	Resolved
H-7	Operator can introduce leverage without consent from all DSSs	Resolved
H-8	Loops are unconstrained which allows for DOS and loss of funds	Open
H-9	A malicious DSS can intentionally DOS operators	Open
M-1	Operator can call `Core.finalizeUpdateStakeInDSS()` after unregistering from the DSS	Resolved
M-2	`Core.finishRedeem()`: Redemptions cannot be finished by anyone due to allowance check	Resolved
M-3	Custom `ERC4626` logic breaks specification	Open
M-4	`SlashingHandler.sol` is incompatible with tokens that revert on transfer to 0x such as stETH	Resolved
M-5	Low-level `call` to DSS does not behave as expected which breaks considerations around gas consumption	Open
L-1	IDSS.unregister() function is never called	Resolved
L-2	CanceledStakeUpdate event is emitted with wrong parameter	Resolved
L-3	`CoreLib.createVault()` can set the vault implementation directly, without intializing against the Default implementation	Resolved
L-4	Operator can use a custom DSS to frontrun a self-slashing, avoiding slashing from a honest DSS	Acknowledged
L-5	`SlasherLib.QueuedSlashing` should include a nonce to avoid identical slash requests in the same block	Resolved
L-6	A blacklisted staker might leave a redemption open forever	Resolved
L-7	Unspent allowance by `SlashingHandler.sol` might brick slashing	Resolved
L-8	Ambiguity in the upgrade pattern of `Core.sol`	Resolved
I-1	Vault.finishRedeem(): Check that withdrawal exists	Resolved

ID	Description	Status
I-2	Core.cancelUpdateStakeInDSS(): Check if update stake request exists	Resolved
I-3	Operator.sol: Remove unused Signature struct	Resolved
I-4	`IVault` should be updated and inherited by `Vault`	Resolved
I-5	`Core`: Using `onlyOperatorRegisteredToDSS` and `onlyOperator` modifier is redundant	Resolved

4 Findings

High Risk

[H-1] Redemptions in Vault can be cancelled by anyone

Context:

- Vault.sol#L167-L177

Description: The `Vault.cancelRedeem()` function must check that `msg.sender` is the `staker` that has requested the withdrawal. Without this check, anyone can cancel the withdrawal of anyone else.

Recommendation:

```
     function cancelRedeem(bytes32 withdrawalKey) external nonReentrant whenNotPaused
     {
         VaultLib.State storage state = _state();
         WithdrawLib.QueuedWithdrawal storage startedWithdrawal =
         state.withdrawalMap[withdrawalKey];
-        if (startedWithdrawal.start == 0) {
+        if (startedWithdrawal.start == 0 || msg.sender != startedWithdrawal.staker) {
             revert WithdrawalNotFound();
         }
```

Karak: Fixed

Renascence: Fixed by removing `cancelRedeem()`. Redemptions can no longer be cancelled as a result of fixing other issues.

[H-2] Percentage based allocations introduce race conditions and DSS can slash more funds than operator has staked to it

Context:

- Core.sol#L319-L343
- Core.sol#L370-L391

Description: Each `DSS` gets assigned a percentage of the funds in the `Vault`. When there occurs a slashing, the `DSS` yet again has to specify a percentage of the percentage to slash. For example, if the `DSS` is assigned 50% of the funds in the `Vault` and slashes 50% of its funds, it will be `50%*50%=25%` of the funds in the `Vault`.

The problem is that after a slashing operation, the `vaultStakes` are not reduced. That means that after the slashing of 50% of the DSS's funds, it's percentage is still registered as 50%.

This allows the DSS to perform the slashing again based on the assumption it has 50% of funds in the `Vault` staked to it.

The scenario can be verified with the following test case:

```
function test_slash_operator_twice() public {
    uint96 slashPercentageWad = uint96(Constants.HUNDRED_PERCENT_WAD);
```

```
        add_stake_to_dss();
        uint96[] memory slashPercentagesWad = new uint96[](2);
        slashPercentagesWad[0] = slashPercentageWad;
        slashPercentagesWad[1] = slashPercentageWad;

        vm.startPrank(address(dss));
        (address[] memory operatorVaults, uint256[] memory initStakes) =
        core.fetchVaultStakesInDSS(operator, dss);
        uint256[] memory initialAssets = compute_initial_assets(operatorVaults);

        SlasherLib.SlashRequest memory slashingReq = SlasherLib.SlashRequest({
            operator: operator,
            slashPercentagesWad: slashPercentagesWad,
            vaults: operatorVaults
        });

        SlasherLib.QueuedSlashing memory queuedSlashing =
        core.requestSlashing(slashingReq);
        vm.warp(block.timestamp + 1);
        SlasherLib.QueuedSlashing memory queuedSlashing2 =
        core.requestSlashing(slashingReq);

        vm.warp(block.timestamp + Constants.SLASHING_VETO_WINDOW);

        console.log(initStakes[0]);
        console.log(initStakes[1]);
        console.log(Vault(operatorVaults[0]).totalAssets());
        console.log(Vault(operatorVaults[1]).totalAssets());

        console.log(queuedSlashing.earmarkedStakes[0]);
        console.log(queuedSlashing.earmarkedStakes[1]);

        core.finalizeSlashing(queuedSlashing);
        core.finalizeSlashing(queuedSlashing2);

        /* @audit-issue
        initStakes are not updated after the slashing
        */
        (operatorVaults, initStakes) = core.fetchVaultStakesInDSS(operator, dss);
        console.log(initStakes[0]);
        console.log(initStakes[1]);
        /* @audit-issue
        DSS can slash more than 100% of its assets
        */
        console.log(Vault(operatorVaults[0]).totalAssets());
        console.log(Vault(operatorVaults[1]).totalAssets());
    }
```

By repeatedly slashing, the amount of funds that each `DSS` can slash approaches 100% instead of the initial percentage of stakes assigned to it.

Recommendation: A straightforward solution appears to be to lower the percentage of funds allocated to the `DSS` after the slashing is performed. At the same time, the allocation to all other `DSS`'s must be increased accordingly, such that they still have access to the same amount of funds in absolute terms.

Implementing this solution fails to recognize the broader flaw that percentage-based `Vault` allocations introduce race conditions between different `DSS`'s and `DSS/operator`. For example, since slashings can also be cancelled, a `DSS` that requests a slashing can never specify a percentage and be sure it corresponds to the intended amounts slashed.

Hence, the broader recommendation is to assess percentage-based allocations and to determine if it's conceptually possible to make the approach work. Only then can the code be changed accordingly.

Karak: Fixed in different commits, all changes applied by this commit.

Renascence: The issue described here originates from and impacts different parts of the codebase. In addition, slashing depends on the specific DSS implementations which are not implemented yet. As such, the slashing mechanism cannot be fully audited and resolved at this point. However, it is possible to describe the new mechanism and to assess whether the concerns in this report have been addressed.

Slashings are requested by the DSS as a percentage (`slashPercentagesWad`). The percentage is then converted to an absolute amount of assets based on `totalAssets()` of the Vault that is slashed.

```
for (uint256 i = 0; i < stakes.length; ++i) {
    uint256 netSlashWAD =
        Math.mulDiv(stakes[i], slashingMetadata.slashPercentagesWad[i],
        Constants.MAX_SLASHING_PERCENT_WAD);
    earmarkedStakes[i] = Math.mulDiv(
        netSlashWAD, IVault(slashingMetadata.vaults[i]).totalAssets(),
        Constants.MAX_SLASHING_PERCENT_WAD
    );
}
queuedSlashing = QueuedSlashing({
    dss: dss,
    timestamp: uint96(block.timestamp),
    operator: slashingMetadata.operator,
    vaults: slashingMetadata.vaults,
    earmarkedStakes: earmarkedStakes,
    nonce: nonce
});
```

Previously, slashings were queued as a percentage, not as an absolute amount.

After the queue period, the absolute amount in the `QueuedSlashing` is then slashed, the amount is capped at `totalAssets()` of the Vault.

```
function slashAssets(uint256 totalAssetsToSlash, address slashingHandler)
        external
        onlyCore
        returns (uint256 transferAmount)
    {
        transferAmount = Math.min(totalAssets(), totalAssetsToSlash);

        // Approve to the handler and then call the handler which will draw the funds
        SafeTransferLib.safeApproveWithRetry(asset(), slashingHandler,
        transferAmount);
        ISlashingHandler(slashingHandler).handleSlashing(IERC20(asset()),
        transferAmount);

        emit Slashed(transferAmount);
    }
```

After the slashing, the DSS's stake in the Vault is updated:

```
uint256 postSlashingStake = (slashedAssets >= currentSlashableAssets)
    ? 0
    : (
        Math.mulDiv(
            Constants.MAX_SLASHING_PERCENT_WAD,
            (currentSlashableAssets - slashedAssets),
            IVault(queuedSlashing.vaults[i]).totalAssets()
        )
    );
self.operatorState[queuedSlashing.operator].updateStakeInDSS(
    address(queuedSlashing.vaults[i]), queuedSlashing.dss, postSlashingStake
);
```

This mechanism still leaves open concerns:

- What happens if the slashing request is front-run such that the `slashPercentagesWad` does not correspond to the desired amount?
- Does the accounting break when the DSS has queued more than one slashing per Vault? Consider the following test that shows that the DSS is still able to slash more assets than have been assigned to it:

```
function test_slash_operator_twice() public {
        uint96 slashPercentageWad = uint96(Constants.HUNDRED_PERCENT_WAD);
        /* @audit-info
        assigns 500 assets in Vault0 to dss
        */
        add_stake_to_dss();
        uint256 newStakeDSS2 = Constants.HUNDRED_PERCENT_WAD / 2;

        vm.prank(operator);
        core.registerOperatorToDSS(dss2, "");
        // update_vault_stake_to_dss(address(vaults[0]), newStakeDSS2, dss2);
        uint96[] memory slashPercentagesWad = new uint96[](vaults.length);
        slashPercentagesWad[0] = slashPercentageWad;
        slashPercentagesWad[1] = slashPercentageWad;

        vm.startPrank(address(dss));
        (address[] memory operatorVaults, uint256[] memory initStakes) =
        core.fetchVaultStakesInDSS(operator, dss);

        SlasherLib.SlashRequest memory slashingReq = SlasherLib.SlashRequest({
            operator: operator,
            slashPercentagesWad: slashPercentagesWad,
            vaults: operatorVaults
        });

        SlasherLib.QueuedSlashing memory queuedSlashing =
        core.requestSlashing(slashingReq);

        vm.warp(block.timestamp + 1);

        uint96[] memory slashPercentagesWad2 = new uint96[](vaults.length);
        slashPercentagesWad2[0] = slashPercentageWad;
        slashPercentagesWad2[1] = slashPercentageWad;
        SlasherLib.SlashRequest memory slashingReq2 = SlasherLib.SlashRequest({
            operator: operator,
            slashPercentagesWad: slashPercentagesWad2,
            vaults: operatorVaults
        });

        SlasherLib.QueuedSlashing memory queuedSlashing2 =
        core.requestSlashing(slashingReq2);

        vm.warp(block.timestamp + Constants.SLASHING_VETO_WINDOW);

        core.finalizeSlashing(queuedSlashing);
        core.finalizeSlashing(queuedSlashing2);

        /* @audit-issue
        The DSS only had 500 assets assigned but totalAssets() in the Vault is zero
        -> DSS was able to slash twice its assets
        */
        console.log("assets in vault", IVault(operatorVaults[0]).totalAssets());
        assertEq(IVault(operatorVaults[0]).totalAssets(), 0);
    }
```

[H-3] `Core`: DSS is called with wrong hook functions

Context:

- Core.sol#L336

Description: In `Core.requestSlashing()`, the `finishSlashingHook()` function is called instead of the correct `requestSlashingHook()` function.

Recommendation: Note that `IDSS` defines the `requestSlashingHook()` function with a `uint256 slashingPercentageWad` parameter, whereas the `slashingRequest` contains an array of `uint256` with slashing percentages. So the `requestSlashingHook()` function must also accept an array of `uint256` as parameter.

```diff
         HookLib.callHookIfInterfaceImplemented(
             dss,
-            abi.encodeWithSelector(dss.finishSlashingHook.selector,
slashingRequest.operator),
+            abi.encodeWithSelector(
+                dss.requestSlashingHook.selector, slashingRequest.operator,
slashingRequest.slashPercentagesWad
+            ),
             Constants.DSS_START_UPDATE_STAKE_ID,
             true,
             Constants.DEFAULT_HOOK_GAS
```

```diff
     function requestUpdateStakeHook(address operator, Operator.StakeUpdateRequest
     memory newStake) external;
     function cancelUpdateStakeHook(address operator, address vault) external;
     function finishUpdateStakeHook(address operator) external;
-    function requestSlashingHook(address operator, uint256 slashingPercentageWad)
external;
+    function requestSlashingHook(address operator, uint256[] memory
slashPercentagesWad) external;
     function cancelSlashingHook(address operator) external;
     function finishSlashingHook(address operator) external;
```

It's also possibly useful to include the `vaults` that are slashed as a parameter.

Karak: Fixed

Renascance: Fixed as recommended.

[H-4] `Core`: `callHookIfInterfaceImplemented()` **is called with wrong interface IDs**

Context:

- Core.sol#L28-L519

Description: In `Core`, there are multiple instances where `callHookIfInterfaceImplemented()` is called with the wrong `interfaceId` parameter. As a consequence, the hook won't be called if it is supported and it will be called if it is not supported.

Recommendation:

```diff
@@ -122,7 +122,7 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
         HookLib.callHookIfInterfaceImplemented(
             dss,
             abi.encodeWithSelector(dss.registrationHook.selector, operator,
             registrationHookData),
-            Constants.DSS_UNREGISTRATION_ID,
+            Constants.DSS_REGISTRATION_ID,
             true, // So it can't prevent the operator from unregistering
             Constants.DEFAULT_HOOK_GAS
         );
@@ -156,7 +156,7 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
         HookLib.callHookIfInterfaceImplemented(
             dss,
             abi.encodeWithSelector(dss.unregistrationHook.selector, operator,
             unregistrationHookData),
-            Constants.DSS_REGISTRATION_ID,
+            Constants.DSS_UNREGISTRATION_ID,
             false,
             Constants.DEFAULT_HOOK_GAS
         );
@@ -216,7 +216,7 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
         HookLib.callHookIfInterfaceImplemented(
             dss,
             abi.encodeWithSelector(dss.cancelUpdateStakeHook.selector, operator,
             vault),
-            Constants.DSS_START_UPDATE_STAKE_ID,
+            Constants.DSS_CANCEL_UPDATE_STAKE_ID,
             true,
             Constants.DEFAULT_HOOK_GAS
         );
@@ -241,7 +241,7 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
         HookLib.callHookIfInterfaceImplemented(
             dss,
             abi.encodeWithSelector(dss.finishUpdateStakeHook.selector, msg.sender),
-            Constants.DSS_START_UPDATE_STAKE_ID,
+            Constants.DSS_FINISH_UPDATE_STAKE_ID,
             true,
             Constants.DEFAULT_HOOK_GAS
         );
@@ -334,7 +334,7 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
         HookLib.callHookIfInterfaceImplemented(
             dss,
             abi.encodeWithSelector(dss.finishSlashingHook.selector,
             slashingRequest.operator),
-               Constants.DSS_START_UPDATE_STAKE_ID,
+               Constants.DSS_REQUEST_SLASHING_ID,
             true,
             Constants.DEFAULT_HOOK_GAS
         );
@@ -356,7 +356,7 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
         HookLib.callHookIfInterfaceImplemented(
             dss,
             abi.encodeWithSelector(dss.cancelSlashingHook.selector,
             queuedSlashing.operator),
-               Constants.DSS_START_UPDATE_STAKE_ID,
+               Constants.DSS_CANCEL_SLASHING_ID,
             true,
             Constants.DEFAULT_HOOK_GAS
         );
@@ -382,7 +382,7 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
         HookLib.callHookIfInterfaceImplemented(
             dss,
             abi.encodeWithSelector(dss.finishSlashingHook.selector,
             queuedSlashing.operator),
-               Constants.DSS_START_UPDATE_STAKE_ID,
+               Constants.DSS_FINISH_SLASHING_ID,
             true,
             Constants.DEFAULT_HOOK_GAS
         );
```

Karak: Fixed

Renascence: Hook interface IDs are no longer identified by the same IDs. Now, they are identified by the function selectors. In all instances, the correct function signatures are used, and so the issue is fixed.

[H-5] `Core`: `callHookIfInterfaceImplemented()` is called with wrong `canFail` parameters

Context:

- Core.sol#L126
- Core.sol#L160

Description: Providing the wrong `canFail` parameter allows the DSS to block execution when it shouldn't be able to and doesn't allow it to block execution when it should be able to.

Recommendation: Change the `canFail` parameter to `false` here, and change it to `true` here.

Karak: Fixed

Renascence: Fixed as recommended.

[H-6] `Vault`: Redemption process can be gamed to avoid slashing

Context:

- Vault.sol#L109-L163

Description: The protocol expects to implement timing constraints such that a staker, becoming aware of an upcoming slashing, is not able to leave the protocol and avoid the loss. However, the redemption process is flawed and must be redesigned. By applying the following strategy, a staker can always avoid slashing.

1. `account1` calls `Vault.startRedeem()` for some amount `x` of shares
2. The `account1` that calls `Vault.startRedeem()` is not the account that actually will hold the shares to be redeemed
3. `account1` waits for withdrawal delay to pass
4. `account2` deposits into the `Vault`
5. Whenever there is a slashing, `account2` transfers their shares to `account1` and `account1` calls `Vault.finishRedeem()`

Recommendation: To prevent this issue, and to harden the `Vault` against other attempts to game the redemption process, the following steps should be considered:

1. When a user calls `Vault.startRedeem()` the shares are transferred to `Vault`, and the user cannot transfer them anymore
2. User cannot cancel the redemption
3. After the withdrawal delay, anyone can finish the redemption. There must not be a condition under which redemption can fail and the user must not be able to get back their shares

Karak: Fixed

Renascence: The liquidation flow has been refactored as recommended.

[H-7] Operator can introduce leverage without consent from all DSSs

Context:

- Core.sol#L171-L197

Description: In the `Core.requestUpdateStakeInDSS()` function, the `requestUpdateStakeHook` is called such that the DSS for which the stake is increased is able to decline the request by reverting. The problem is that this DSS may accept leverage, whereas other DSSs that also have a stake assigned do not. The other DSSs do not have to consent to the stake increase of the new DSS even though their own security is affected by it.

Recommendation: There must be a mechanism by which all DSSs with a stake allocated must consent to a stake increase for any other DSS.

Karak: Our approach to this issue is two fold:

- add a function to get the leverage of a vault to a DSS
- let the DSS do a `jail` inside it's own DSS contract with whatever arbitrary logic but the reference would check if the leverage is above 100% and allow a DSS to `jail` them. `Jail` doesn't

mean they are kicked from the set but instead their allocations and responses aren't considered by the DSS until the DSS `unjails` them

Renascence: The solution for this issue is based on the DSS implementation. It is necessary that the DSSs are audited and checked for compatibility with the Core contract.

An alternative recommendation is to enforce that the sum of a Vault's stakes towards DSSs does not exceed 100%.

[H-8] Loops are unconstrained which allows for DOS and loss of funds

Context:

- Karak-Restaking

Description: The issue has been reported by the client at the start of the audit.

In the protocol, there exist several unbounded loops where the iterations can be determined by one of the actors, e.g., by creating additional Vaults.

At the start of the audit, the client has still been working on a fix for this, so the finding should serve as a reminder that the issue must be addressed.

Recommendation: A possible approach is to limit the loops by limiting how many items can be created that must be iterated over, e.g., how many Vaults an operator can create

Karak: Fixed

Renascence: An operator is now limited to creating 32 Vaults. And a slashing request is limited to the same number of Vaults.

However, the new `getVaultLeverage()` function now iterates over all DSSs that an operator is registered to which does not have a maximum amount.

[H-9] A malicious DSS can intentionally DOS operators

Context:

- HookLib.sol

Description: The `callHookIfInterfaceImplemented()` function is called at the end of any operator related call in `Core.sol` such as `requestUpdateStakeInDSS()`, `cancelUpdateStakeInDSS()`, etc.

```
function callHookIfInterfaceImplemented(IDSS dss, bytes memory data, bytes4
interfaceId, bool canFail, uint256 gas)
    internal
    returns (bool)
{
    if (!dss.supportsInterface(interfaceId)) {
        return false;
    }
    return callHook(dss, data, canFail, gas);
}
```

However, the protocol has no control over the implementation of the DSS, e.g., it can be an upgradeable contract that can change its functionality, or it can just intentionally remove the function selector from its implementation to DOS any operators associated with it.

The vulnerability applies to all the functions below:

```
function registrationHook(address operator, bytes memory extraData) external;
function unregistrationHook(address operator, bytes memory extraData) external;

function requestUpdateStakeHook(address operator, Operator.StakeUpdateRequest memory
newStake) external;
function cancelUpdateStakeHook(address operator, address vault) external;
function finishUpdateStakeHook(address operator) external;
function requestSlashingHook(address operator, uint256 slashingPercentageWad)
external;
function cancelSlashingHook(address operator) external;
function finishSlashingHook(address operator) external;
```

Recommendation: The DSS should not be able to halt the execution flow in `dss.supportsInterface()`. A good approach might be to check if the call has failed, emit an event with the return data (limited in size), and process the result off-chain. The same approach can be used that was recommended for the low-level call in `callHook()`.

Karak: Fixed

Renascence: The issue still exists. `dss.supportsInterface()` must be changed into a low-level call with a fixed gas amount and fixed return data size, like it is implemented in `HookLib.callHook()`.

```
     ) internal returns (bool) {
-        if (!dss.supportsInterface(interfaceId)) {
+        bytes memory interfaceData =
abi.encodeWithSelector(IERC165.supportsInterface.selector, interfaceId);
+
+        bool returnData;
+        bool success;
+
+        uint256 returnDataSize;
+
+        assembly {
+            let freeMemPointer := mload(0x40) // Get free memory pointer
+
+            success :=
+                call(
+                    50000, // gas available to the inner call
+                    dss, // address of contract being called
+                    0, // ETH (denominated in WEI) being transferred in this call
+                    add(interfaceData, 0x20), // Pointer to actual data (i.e. 32
bytes offset from `data`)
+                    mload(interfaceData), // Size of actual data (i.e. the value
stored in the first 32 bytes at `data`)
+                    freeMemPointer, // Free pointer as a buffer for the inner call to
write the return value
+                    32 // 32 bytes size limit for the return value
+                )
+
+            returnData := mload(freeMemPointer)
+        }
+        if (!success || !returnData) {
            emit InterfaceNotSupported();
+            return false;
```

```
        }
        return callHook(address(dss), data, ignoreFailure, gas);
    }
```

Medium Risk

[M-1] Operator can call `Core.finalizeUpdateStakeInDSS()` after unregistering from the DSS

Context:

- Core.sol

Description: `Core.requestUpdateStakeInDSS()` has a `onlyOperatorRegisteredToDSS` modifier that allows only operators registered with the requested DSS to request stake updates. The issue is that an operator can call `requestUpdateStakeInDSS()`, unregister from the DSS by calling `Core.unregisterOperatorFromDSS()`, and finally call `Core.finalizeUpdateStakeInDSS()`, which passes `canFail = true` to `HookLib.callHookIfInterfaceImplemented()`. Even if the DSS reverts, the Operator state `State.vaultStakes` would still record an update that there are staked funds towards the DSS for a particular vault that belongs to the operator. If the DSS does rely on `Core.sol` to verify that operators are registered with it and `HookLib.callHookIfInterfaceImplemented()` doesn't revert, then the DSS won't be able to use `Core.requestSlashing()`, since it requires the operator to be registered with the DSS.

Recommendation: If every DSS ensures its own access control that the operator calling `Core.finalizeUpdateStakeInDSS()` is registered with the DSS, then `Core.finalizeUpdateStakeInDSS()` should set `canFail = false` to avoid recording an incorrect state in `Core.sol`. In the case where DSSs rely on `Core.sol` to perform an access control check on the operator, `Core.finalizeUpdateStakeInDSS()` must have an `onlyOperatorRegisteredToDSS` modifier.

```
@@ -230,10 +231,11 @@ contract Core is IBeacon, ICore, OwnableRoles, Initializable,
ReentrancyGuard, U
     function finalizeUpdateStakeInDSS(Operator.QueuedStakeUpdate memory queuedStake)
         external
         nonReentrant
         whenNotPaused
         onlyOperator(msg.sender)
+        onlyOperatorRegisteredToDSS(msg.sender, queuedStake.updateRequest.dss)
     {
```

Karak: Fixed

Renascence: Fixed as recommended.

[M-2] `Core.finishRedeem()`: Redemptions cannot be finished by anyone due to allowance check

Context:

- Vault.sol#L128

Description: In `Vault`, it is stated that anyone can finish another user's redemption. But this is not possible, since `ERC4626._withdraw()` checks that `msg.sender` has sufficient allowance if `msg.sender != startedWithdrawal.staker`.

In addition to the unexpected behavior, the fact that the staker can prevent other users from finishing their withdrawal, has consequences in terms of the strategies that a staker can use to maximize his profit. The fact that redemptions are broken from this point of view is however dealt with in another finding.

Recommendation: `Vault.finishRedeem()` must not use `ERC4626._withdraw()`. Instead it should directly implement the `_withdraw()` code without the allowance check.

Karak: Fixed

Renascence: By refactoring the redemption flow, shares for queued redemptions are now held by the `Vault`, and in `finishRedeem()` `_withdraw()` is called with `by = address(this)` and `owner = address(this)`. As a result, no allowance from the `staker` to `msg.sender` is needed anymore.

[M-3] Custom `ERC4626` logic breaks specification

Context:

- Vault.sol#L24-L306

Description: `Vault` inherits from `ERC4626` and modifies its functionality. The `redeem()` and `withdraw()` functions always revert and redemptions are a two step process handled by `startRedeem()` and `finishRedeem()`. This is not according to the ERC4626 specification, e.g., `previewWithdraw()` and `previewRedeem()` will return wrong results since the `withdraw()` and `redeem()` function are not available.

Recommendation: If compatibility with `ERC4625` is intended, the `Vault` must be changed in a way that is compatible with the specification. if compliance is not needed, the finding can be acknowledged.

Karak: Fixed

Renascence: The issue has been marked as fixed in the above Pull Request. However, what has been done is to replace calls to `previewRedeem()` with `convertToAssets()`. This is an optimization but does not have implications on ERC4626 compliance.

The `Vault` implements a custom redemption process with `startRedeem()` and `finishRedeem()` but ERC4626 expects that withdrawals and redemptions adhere to ERC4626.

Note that the EIP specification also refers to the case of a two step withdrawal process:

> Note that some implementations will require pre-requesting to the Vault before a withdrawal may be performed. Those methods should be performed separately.

So the redemption request can be made with `startRedeem()` but finishing the withdrawal must be performed with the regular `redeem()` function that adheres to the ERC4626 specification.

[M-4] `SlashingHandler.sol` **is incompatible with tokens that revert on transfer to 0x such as stETH**

Context:

- SlashingHandler.sol

Description: The current implementation of `SlashingHandler` handles slashing by first transferring the asset from the Vault and then transferring the asset to the zero address with the intent of burning it.

```
function handleSlashing(IERC20 token, uint256 amount) external {
    if (amount == 0) revert ZeroAmount();
    if (!_config().supportedAssets[token]) revert UnsupportedAsset();

    SafeTransferLib.safeTransferFrom(address(token), msg.sender, address(this),
    amount);

    SafeTransferLib.safeTransfer(address(token), address(0), amount);
}
```

However, that won't work for many tokens (e.g., OZ's ERC20) because they revert when attempting to transfer to address(0).

A great example is `stETH`. Its `transfer` function calls `_transferShares`, which will revert if the address is 0x.

Another point to consider is that anyone can call this function. Hence, there's a real possibility that an attacker uses the contract to perform malicious actions such that the contract gets blacklisted from transfers.

Recommendation: Taking care of the 0x transfer issue can be as simple as ensuring `SlashingHandler.sol` works with a predefined set of assets.

To ensure that no malicious actions can be performed with the handler, you can make sure it's only callable by valid vaults. For this to work you need to keep track of created vaults.

```
    function handleSlashing(IERC20 token, uint256 amount) external {
        if (amount == 0) revert ZeroAmount();
        if (!_config().supportedAssets[token]) revert UnsupportedAsset();

+       require(core.isValidVault(msg.sender))

        SafeTransferLib.safeTransferFrom(address(token), msg.sender, address(this),
        amount);

        SafeTransferLib.safeTransfer(address(token), address(0), amount);
    }
```

Karak: The `SlashingHandler.sol` contract is actually an ancillary contract that is set for a specific asset via the `allowlistAssets()` entrypoint in the `Core.sol` contract. This means that we will likely deploy a set of `SlashingHandler`'s for different type of assets with different slashing logic.

As for a `SlashingHandler` instance getting blacklisted, one of Core's manager roles could just reassign the `SlashingHandler` of a particular asset to another address via the same entry point mentioned above.

Renascence: Based on the above reasoning, specifically that there will be different `SlashingHandler` contracts for different assets, the finding can be considered resolved.

For the `SlashingHandler` that is currently implemented, the blacklisting scenario is not a concern since tokens will just be transferred to `address(0)`, so there's no reason for `SlashingHandler` to get blacklisted. Still, future implementations of `SlashingHandler` must check if it's safe that anyone can call the `handleSlashing()` function and implement access controls if necessary.

[M-5] Low-level `call` to DSS does not behave as expected which breaks considerations around gas consumption

Context:

- HookLib.sol#L8-L13

Description: The `HookLib.callHook()` function performs a low-level `call` into the DSS.

```
function callHook(IDSS dss, bytes memory data, bool canFail, uint256 gas) internal
returns (bool) {
    if (gasleft() < gas) revert NotEnoughGas();
    (bool success, bytes memory returnData) = address(dss).call{gas: gas}(data);
    if (!canFail && !success) revert DSSHookCallReverted(returnData);
    return success;
}
```

There are two assumptions that need to hold:

1. The call to the hook is made with very close to `gas` amount of gas
2. The low-level call cannot cause a revert

Both assumptions have problems. For 1), the low-level `call` is performed with at most 63/64th of the gas that the caller context has available. So to ensure that the low-level `call` is performed with at least `gas`, `gasleft()` must be greater than `gas * 64 / 63 + buffer`. `buffer` is the worst case amount of gas that is needed from the check until the DSS is called. It can be determined via testing.

For 2), there is a potential DOS issue by returning a large amount of data in the low-level `call`. However, the return data must be written to memory by the DSS. So, for any reasonable amount of gas passed to the DSS (like 500k) it should not be possible for the DSS to return sufficient amount of data (which has to be written to memory before it can be returned) to cause a DOS in the caller.

The `callHook()` function writes all of the return data to memory with a `RETURNDATACOPY` call that has checked the size of the return data with `RETURNDATASIZE`. To fix this concern, only a fixed length of data should be decoded.

Recommendation: For 1), it is recommended to perform gas benchmarks and to adjust the check accordingly such that the low-level `call` is always performed with sufficient gas.

For 2), a low-level call in assembly must be performed that limits the amount of returndata copied into memory by supplying a `retSize` argument.

Karak: Fixed `gasleft()` check here. Fixed return data size here.

Renascence: The first problem has been fixed by adding the following check:

```
if (gasleft() < (hookGasLimit * 64 / 63 + Constants.HOOK_GAS_BUFFER)) revert
NotEnoughGas();
```

As part of the mitigation review, the gas constraints have been tested and they work correctly. Still, it is recommended that Karak implements such gas tests and adds them to their test suite.

For the second issue, it is recommended to make the assembly code more readable and to avoid low level memory manipulation by applying the following changes:

```
## HookLib.sol

         if (gasleft() < (hookGasLimit * 64 / 63 + Constants.HOOK_GAS_BUFFER)) revert
         NotEnoughGas();

-        bytes memory returnData;
+        bytes32[1] memory returnData;

         assembly {
-            returnData := mload(0x40) // Set returnData to the current free memory
pointer
-
-            // pointer(data) + 0x20 is where actual data is available
-            // pointer(data) contains the size of the data in bytes
-            // returnData is where the return value is written to
-            // we limit size of return value to 32 bytes (same as the size of
`returnData` above)
             success :=
                 call(
                     hookGasLimit, // gas available to the inner call
@@ -32,25 +27,11 @@ library HookLib {
                     add(data, 0x20), // Pointer to actual data (i.e. 32 bytes offset
                     from `data`)
                     mload(data), // Size of actual data (i.e. the value stored in the
                     first 32 bytes at `data`)
                     returnData, // Free pointer as a buffer for the inner call to
                     write the return value
-                    MAX_RETURN_DATA_SIZE // 32 bytes size limit for the return value
+                    32 // 32 bytes size limit for the return value
                 )
-
-            // Check the size of the return data
-            let size := returndatasize()
-
-            // Copy the return data to the allocated memory, limit to
MAX_RETURN_DATA_SIZE
-            if gt(size, MAX_RETURN_DATA_SIZE) { size := MAX_RETURN_DATA_SIZE }
-
-            // Allocate memory for return data
-            mstore(0x40, add(add(returnData, 0x20), size)) // Update free memory
pointer to new end of allocated space
-            mstore(returnData, size) // Store the size of the return data at the
beginning of returnData
-
-             // Copy 32 bytes from the free pointer to `returnData`'s data slot (i.e.
32 bytes offset from base `returnData` pointer)
-             returndatacopy(add(returnData, 0x20), 0, size)
         }
-
-        if (!ignoreFailure && !success) revert DSSHookCallReverted(returnData);
-        emit HookCallFailed(returnData);
+        if (!ignoreFailure && !success) revert DSSHookCallReverted(returnData[0]);
+        emit HookCallFailed(returnData[0]);
         return success;
     }


## Events.sol

-event HookCallFailed(bytes returnData);
+event HookCallFailed(bytes32 returnData);


## Errors.sol

-error DSSHookCallReverted(bytes revertReason);
+error DSSHookCallReverted(bytes32 revertReason);
```

Low Risk

[L-1] IDSS.unregister() function is never called

Context:

- IDSS.sol#L9

Description: The `IDSS` interface defines the functions that every DSS must implement. One of the functions is `unregisterOperator()`. However, the protocol never calls the function. If it is intended to call the function, it should be called in `CoreLib.unregisterOperatorFromDSS()`.

Recommendation: Call `dss.unregisterOperator()` in `CoreLib.unregisterOperatorFromDSS()`. However, the same non-griefing precautions need to be taken as in the `HookLib` library. DSS must not be allowed to revert execution when an operator wants to unregister.

Karak: Removed `unregisterOperator()` from `IDSS`.

Renascence: Fixed. The registration and unregistration hooks that are called from within `Core.sol` are sufficient to notify the DSS. The `IDSS.cancelUpdateStakeHook()` can also be removed from the DSS interface.

[L-2] CanceledStakeUpdate event is emitted with wrong parameter

Context:

- Core.sol#L224

Description: The event is emitted with `stakeHash=self.operatorState[operator].pendingStakeUpdate[vault]`, but `self.operatorState[operator].pendingStakeUpdate[vault]` has already been set to `bytes32(0)`.

Recommendation: Emit the event with the old value of `self.operatorState[operator].pendingStakeUpdate[vault]`.

Karak: Fixed

Renascence: Fixed, the event does no longer exist as a result of removing stake update cancellations.

[L-3] `CoreLib.createVault()` can set the vault implementation directly, without intializing against the Default implementation

Context:

- CoreLib.sol#L113-L115

Description: `CoreLib.createVault()` calls `cloneVault()` before setting the implementation. This means all whitelisted implementations must be compatible with the Default implementation initializer. Setting the implementation before the call to `cloneVault()` will allow for more robust Vault implementations since they won't have to conform to the Default implementation initializer.

Recommendation:

```diff
@@ -41,10 +41,11 @@ library CoreLib {
         mapping(IDSS dss => mapping(address operator => Status)) dssToOperator;
         mapping(address implementation => bool) allowlistedVaultImpl;
         mapping(address asset => address slashingHandler) assetSlashingHandlers;
         mapping(bytes32 slashRoot => bool) slashingRequests;
         address vaultImpl;
+        address pendingImpl;
         uint128 nonce;
         address vetoCommittee;
     }

     function initOrUpdate(Storage storage self, address _vaultImpl, address
     _vetoCommittee) internal {
@@ -108,13 +109,16 @@ library CoreLib {
         address implementation
     ) internal returns (IVault) {
         bytes memory initData = abi.encodeCall(
             IVault.initialize, (address(this), operator, depositToken, vaultType,
             name, symbol, extraData)
         );
+        self.pendingImpl = implementation;
         IVault vault = cloneVault(initData);

         self.vaultToImplMap[address(vault)] = implementation;
+        self.pendingImpl = address(0);
         emit NewVault(address(vault), implementation);
         return vault;
     }
```

```diff
    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 == address(0)) {
+           return self.pendingImpl;
+       } else if (vaultImplOverride == Constants.DEFAULT_VAULT_IMPLEMENTATION_FLAG) {
            return self.vaultImpl;
        }
        return vaultImplOverride;
```

Karak: Fixed

Renascence: Fixed by using `LibClone.predictDeterministicAddressERC1967BeaconProxy()` to predict the address of the new vault.

[L-4] Operator can use a custom DSS to frontrun a self-slashing, avoiding slashing from a honest DSS

Context:

- Core.sol#L110
- Core.sol#L319

Description: An Operator can register a custom DSS and initiate a `Core.requestSlashing()` from the custom DSS in case of slashing by an honest DSS. The Operator can then front-run a `Core.finalizeSlashing()` from their custom DSS. Although the `SlashingHandler` contracts are whitelisted, they can still have custom logic tied to which DSS initiated or finalized the slashing. Depending on the `SlashingHandler` implementation, this procedure could benefit the `Operator` by preventing honest DSSs from finalizing the slashing. Such a front-running scenario is only a concern under the current staking design where an Operator can use leverage and have, for example, a 100% stake in several Vaults.

Recommendation If the staking mechanism is redesigned so that Operators cannot use leverage and their Vault stakes sum up to `<= 100%`, this issue will be mitigated. Alternatively, a solution would be to whitelist DSS providers.

Karak: Since the fake DSS is still slashing the same asset in this scenario and we (the manager role in Core) set the `SlashingHandler` of a particular asset, even if they slash it before the real DSS, the end result is the same.

We acknowledge that in the event the slashing logic isn't simply burning the tokens and instead does something else like, for example, transfer it to the slasher, this allows the to-be-slashed operator to use a fake DSS and escape with the funds. But, this wouldn't happen since we only plan to whitelist SlashingHandler's we write that just burn the tokens.

Renascence: Acknowledged.

[L-5] `SlasherLib.QueuedSlashing` should include a nonce to avoid identical slash requests in the same block

Context:

- Core.sol#L171
- SlasherLib.sol#L48-L55

Description: If a DSS happens to slash the same vault more than once, for identical amounts, in the same block, the `QueuedSlashing` from the first slash request will be overwritten by the subsequent `QueuedSlashing` request, which can then be overwritten by the next request, and so on. This becomes a concern if the DSS implementation relies on `Core.cancelSlashing()` or `Core.finalizeSlashing()` being available to be executed, for every request, to perform state-changing operations in the DSS through the `HookLib` calls.

Recommendation: Include a tracking nonce value in `QueuedSlashing` for every DSS.

Karak: Fixed

Renascence: Fixed as recommended by using a nonce.

[L-6] A blacklisted staker might leave a redemption open forever

Context:

- Vault.sol

Description: When a staker initiates a redemption through `Vault.startRedeem(...)`, they are assigned as the receiver of the queued withdrawal to be processed.

```
>        address staker = msg.sender;

        uint256 assets = super.previewRedeem(shares);

        withdrawalKey = WithdrawLib.calculateWithdrawKey(staker,
        state.stakerToWithdrawNonce[staker]++);

>        state.withdrawalMap[withdrawalKey].staker = staker;
        state.withdrawalMap[withdrawalKey].start = uint96(block.timestamp);
        state.withdrawalMap[withdrawalKey].assets = assets;
        state.withdrawalMap[withdrawalKey].shares = shares;
```

However, an issue will arise when the underlying asset of the vault is a token that has blacklists (e.g., USDT/USDC).

```
    function finishRedeem(bytes32 withdrawalKey) external nonReentrant whenNotPaused {
        (VaultLib.State storage state, VaultLib.Config  storage config) = _storage();
>        WithdrawLib.QueuedWithdrawal memory startedWithdrawal =
state.withdrawalMap[withdrawalKey];
        ...
        _withdraw({
            by: msg.sender,
>            to: startedWithdrawal.staker,
            owner: startedWithdrawal.staker,
            assets: redeemableAssets,
            shares: shares
        });
```

In this case, the user's redemption can never be finished, and even though it can be canceled, that doesn't help as the staker cannot set another address to withdraw it.

As a result, the redemption might be left open forever, and this might break the delays

Recommendation: Given other recommended fixes are applied, there will no longer be any consequences for leaving redemptions open.

Karak: Fixed by adding `beneficiary` to `Vault.startRedeem()`.

Renascence: By refactoring the redemption process, there is no impact to leaving redemptions open. Still, it is possible that `beneficiary` becomes blacklisted while the redemption is queued, therefore making it impossible to finalize. Overall, no further changes are needed, handling the unlikely case of `beneficiary` being blacklisted while a redemption is queued is over-engineering and not needed from a security perspective.

[L-7] Unspent allowance by `SlashingHandler.sol` might brick slashing

Context:

- Vault.sol

Description: `Vault.sol` needs to grant allowance to `SlashingHandler.sol` so it can transfer from the Vault the to-be-slashed assets in question.

```
function slashAssets(uint256 slashPercentageWad, address slashingHandler) external
onlyCore {
    if (slashPercentageWad == 0) revert ZeroAmount();

    uint256 transferAmount = Math.mulDiv(totalAssets(), slashPercentageWad,
    Constants.MAX_SLASHING_PERCENT_WAD);
    if (transferAmount == 0) revert ZeroAmount();

    // Approve to the handler and then call the handler which will draw the funds
    IERC20 underlyingAsset = IERC20(asset());
    underlyingAsset.approve(slashingHandler, transferAmount);
    ISlashingHandler(slashingHandler).handleSlashing(underlyingAsset, transferAmount);

    emit Slashed(transferAmount);
}
```

However, depending on how the slasher is implemented, an unspent allowance may cause a denial of service during the approval calls.

Some tokens (like USDT) do not work when changing the allowance from an existing non-zero allowance value.They must first be approved by zero and then the actual allowance must be approved.

Recommendation:

```
    function slashAssets(uint256 slashPercentageWad, address slashingHandler) external
    onlyCore {
        if (slashPercentageWad == 0) revert ZeroAmount();

        uint256 transferAmount = Math.mulDiv(totalAssets(), slashPercentageWad,
        Constants.MAX_SLASHING_PERCENT_WAD);
        if (transferAmount == 0) revert ZeroAmount();

        // Approve to the handler and then call the handler which will draw the funds
        IERC20 underlyingAsset = IERC20(asset());
+       underlyingAsset.approve(slashingHandler, 0);
        underlyingAsset.approve(slashingHandler, transferAmount);
        ISlashingHandler(slashingHandler).handleSlashing(underlyingAsset,
        transferAmount);

        emit Slashed(transferAmount);
    }
```

Karak: Fixed

Renascence: Fixed as recommended by using Solady's `safeApproveWithRetry()` function.

[L-8] Ambiguity in the upgrade pattern of `Core.sol`

Context:

- Core.sol

Description: Currently, the `deploy.sol` script utilizes Solady's `ERC1967Factory` contract to deploy `Core.sol`. However, `Core.sol` also inherits `UUPSUpgradeable` and overrides the `_authorizeUpgrade()` function with an `onlyOwner` modifier. The issue is that the Core contract can be upgraded only through the `ERC1967Factory` by the admin of the proxy. If the owner of `Core.sol` is different from the admin stored in the factory, the owner won't be able to upgrade `Core.sol`.

Recommendation: `Core.sol` shouldn't inherit `UUPSUpgradeable` and shouldn't override `_authorize-Upgrade()`. Alternatively, `Core.sol` shouldn't be deployed through the `ERC1967Factory`, but behind a proxy that is compatible with `UUPSUpgradeable` such as OZ ERC1967Proxy.

Karak: Fixed

Renascence: Fixed as recommended by removing `UUPSUpgradeable` and using `ERC1967Factory`.

Informational

[I-1] Vault.finishRedeem(): Check that withdrawal exists

Context:

- Vault.sol#L134-L163

Description: There is no check that `startedWithdrawal` actually exists. So an invalid `withdrawalKey` can be provided that returns a `startedWithdrawal` with all values set to zero.

This ends up having no effect but for best practice it should be prevented.

Recommendation:

```
        (VaultLib.State storage state, VaultLib.Config storage config) = _storage();
        WithdrawLib.QueuedWithdrawal memory startedWithdrawal =
        state.withdrawalMap[withdrawalKey];

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

Karak: Fixed

Renascence: Fixed as recommended.

[I-2] Core.cancelUpdateStakeInDSS(): Check if update stake request exists

Context:

- Core.sol#L203-L225

Description: The function should check that `self.operatorState[operator].pendingStakeUpdate[vault] != bytes32(0)`. There is no immediate security impact to cancelling a stake update request that does not exist, but it can lead to unexpected calls to a DSS and unexpected events being emitted.

Recommendation:

```
        address vault = queuedStake.updateRequest.vault;

        CoreLib.Storage storage self = _self();
+       if (self.operatorState[operator].pendingStakeUpdate[vault] == bytes32(0)) {
+           revert InvalidQueuedStakeUpdateInput();
+       }
        self.operatorState[operator].pendingStakeUpdate[vault] = bytes32(0);
        IDSS dss = queuedStake.updateRequest.dss;
```

Karak: Fixed

Renascence: Fixed as a result of removing `cancelUpdateStakeInDSS()`.

[I-3] Operator.sol: Remove unused Signature struct

Context:

- Operator.sol#L38-L41

Description: The `Signature` struct is never used.

Recommendation: Remove the unused struct.

Karak: Fixed

Renascence: Fixed as recommended.

[I-4] `IVault` should be updated and inherited by `Vault`

Context:

- IVault.sol#L17-L54

Description: `Vault` does currently not inherit from `IVault` since `IVault` specifies some functions like `updateMinWithdrawDelay()` that `Vault` does not implement.

Recommendation: It is recommended that `Vault` inherits `IVault`, and `IVault` should be updated with the up-to-date functions that a Vault is expected to implement.

Karak: Fixed

Renascence: Fixed as recommended.

[I-5] `Core`: Using `onlyOperatorRegisteredToDSS` and `onlyOperator` modifier is redundant

Context:

- Core.sol#L141-L142
- Core.sol#L175-L176

Description: The `onlyOperatorRegisteredToDSS` modifier already contains the `onlyOperator` check. So applying both modifiers to a function is redundant.

Recommendation: Remove the `onlyOperator` modifier in the two affected instances.

Karak: Fixed

Renascence: Fixed as recommended.

4.1 Centralization Risks

4.1.1 The `owner` of `Core.sol` must be trusted

The `owner` of `Core.sol` can use `Core.changeImplementationForVault()` and `Core.changeStandard-Implementation()` to upgrade the implementation of any Vault to an arbitrary one, therefore, acquiring access to all of the user's funds. The `owner` can also pause any Vault for an arbitrary amount of time.

The `admin` of the `Core` Proxy, assigned in Solady `ERC1967Factory.sol`, can upgrade the implementation of `Core` to an arbitrary one, allowing for arbitrary calls to `Vault.slashAssets()`, passing an arbitrary slashing handler. The rights to upgrade the implementation of `Core` also means the `admin` can upgrade the implementation of all Vaults, thereby, acquiring access to all of the user's funds.

4.1.2 The party in control of `VETO_COMMITTEE_ROLE` must be trusted by Operators and DSSs

DSSs must trust the `VETO_COMMITTEE_ROLE` to not maliciously collude with Operators by canceling honest slashing requests. Operators must trust the `VETO_COMMITTEE_ROLE` to cancel any dishonest slashing requests.

4.1.3 The party in control of `MANAGER_ROLE` must be trusted by Operators and DSSs

The `MANAGER_ROLE` can indefinitely pause `Core.sol` and any Vault. Operators and DSSs must trust the `MANAGER_ROLE` to pause only for maintenance or emergency situations.

4.1.4 The `owner` of `SlashingHandler` must be trusted by DSSs

The `owner` of `SlashingHandler` can upgrade the contract to an arbitrary implementation. Therefore, DSSs must trust the `owner` since slashed funds are routed through a `SlashingHandler`.

4.1.5 Users must trust Operators

End users that stake into an Operator's Vaults must trust the Operator. A malicious Operator could lose all the funds staked in his vaults through DSSs slashing the Vaults.

4.1.6 Operators must trust DSS

Operators must trust DSSs to ensure that slashing will only be initiated in cases where the Operators don't meet the performance expectations of the DSS.

4.2 Systemic Risks

4.2.1 Risk of Vaults assets

Users in the Karak Restaking protocol deposit funds in Vaults with whitelisted assets. These assets are external tokens and their risks must be assessed individually and separately from the Karak Restaking protocol.

4.2.2 Distributed Secure Services (DSS) introduce their systemic risks

Even if DSSs behave honestly, each DSS is a separate application with its own systemic risks. Since operators can register to any DSS, it is important that users understand the risks of the DSSs that they are exposed to.

4.2.3 Operators introduce their systemic risks

Operators run the infrastructure to perform tasks for the DSSs. Even if they are trusted entities, the systemic risks in their infrastructure can lead to loss of user funds.