SPEARBIT

# Kiln DeFi integrations v1.2 Security Review

**Auditors**

Noah Marconi, Lead Security Researcher

Akshay Srivastav, Security Researcher

Optimum, Lead Security Researcher

**Report prepared by:** Lucas Goiriz

January 20, 2025

# Contents

# 1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

# 2 Introduction

Kiln is a staking platform you can use to stake directly, or whitelabel staking into your product. It enables users to stake crypto assets, manually or programmatically, while maintaining custody of your funds in your existing solution, such Fireblocks, Copper, or Ledger.

*Disclaimer*: This security review does not guarantee against a hack. It is a snapshot in time of Kiln DeFi integrations v1.2 according to the specific commit. Any modifications to the code will require a new security review.

# 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 |

## 3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

## 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

## 3.3 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

# 4 Executive Summary

Over the course of 5 days in total, Kiln engaged with Spearbit to review the kiln-defi-integrations-v1.2 protocol. In this period of time a total of **23** issues were found.

**Summary**

| | |
|---|---|
| **Project Name** | Kiln |
| **Repository** | kiln-defi-integrations-v1.2 |
| **Commit** | 32b74976 |
| **Type of Project** | Liquid Staking, Enterprise Grade Staking |
| **Audit Timeline** | Dec 3rd to Dec 8th |
| **Fix period** | Dec 16 - Dec 18 |

**Issues Found**

| **Severity** | **Count** | **Fixed** | **Acknowledged** |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 0 | 0 | 0 |
| Medium Risk | 2 | 2 | 0 |
| Low Risk | 8 | 4 | 4 |
| Gas Optimizations | 2 | 1 | 1 |
| Informational | 11 | 3 | 8 |
| **Total** | **23** | **10** | **13** |

# 5 Findings

## 5.1 Medium Risk

### 5.1.1 `VaultUpgradeableBeacon`: The pauser account can unpause or decrease the pause timestamp

**Severity:** Medium Risk

**Context:** ConnectorRegistry.sol#L236, VaultUpgradeableBeacon.sol#L167

**Description:** The `VaultUpgradeableBeacon` contract contains a `pauseFor` function which looks like this:

```
function pauseFor(uint256 duration) external onlyRole(PAUSER_ROLE) {
    if (duration == 0) revert AmountZero();

    uint256 _newPauseTimestamp = block.timestamp + duration;
    if (_newPauseTimestamp <= pauseTimestamp) {
        revert InvalidDuration(_newPauseTimestamp, pauseTimestamp);
    }

    pauseTimestamp = uint88(_newPauseTimestamp);                          // @audit unsafe typecast
    emit Paused(_newPauseTimestamp);
}
```

It can be observed that the `pauseFor` function performs an unsafe `uint88` typecast on a `uint256 _newPauseTimestamp` variable. This unsafe typecasting is not protected against integer overflow and can lead to unintended behaviours.

*Note: `ConnectorRegistry.pauseFor` function also contains the same bug.*

**Impact:** The `PAUSER_ROLE` account can exploit this bug which can lead to two scenarios:

1. The `PAUSER_ROLE` account can unpause the contract.
2. The `PAUSER_ROLE` account can decrease the current `pauseTimestamp` value.

Both of these scenarios are explicitly mentioned to be not allowed in the natspec comments.

**Proof of Concept:**

```
function test_VaultUpgradeableBeacon_pauserCanUnpause() public {
    vm.warp(1000 seconds);
    address pauser = makeAddr("pauser");
    address account = makeAddr("account");
    address impl = address(new FeeDispatcher());
    VaultUpgradeableBeacon beacon = new VaultUpgradeableBeacon(impl, account, account, pauser, account,
↪   account, 100 seconds);

    vm.startPrank(pauser);
    beacon.pause();
    assertEq(beacon.paused(), true);
    beacon.pauseFor(uint256(type(uint88).max) + 1);
    assertEq(beacon.paused(), false);
    vm.stopPrank();
}
function test_VaultUpgradeableBeacon_pauserCanDecreasePauseTS() public {
    vm.warp(1000 seconds);
    address pauser = makeAddr("pauser");
    address account = makeAddr("account");
    address impl = address(new FeeDispatcher());
    VaultUpgradeableBeacon beacon = new VaultUpgradeableBeacon(impl, account, account, pauser, account,
↪   account, 100 seconds);

    vm.startPrank(pauser);
    beacon.pauseFor(100 seconds);
    assertEq(beacon.pauseTimestamp(), 1100 seconds);
    beacon.pauseFor(uint256(type(uint88).max) + 1 + 50 seconds);
    assertEq(beacon.pauseTimestamp(), 1050 seconds);
    vm.stopPrank();
}
```

**Recommendation:** Use Openzeppelin's `SafeCast` library for typecasting.

```
  function pauseFor(uint256 duration) external onlyRole(PAUSER_ROLE) {
      if (duration == 0) revert AmountZero();

      uint256 _newPauseTimestamp = block.timestamp + duration;
      if (_newPauseTimestamp <= pauseTimestamp) {
          revert InvalidDuration(_newPauseTimestamp, pauseTimestamp);
      }

-     pauseTimestamp = uint88(_newPauseTimestamp);
+     pauseTimestamp = SafeCast.toUint88(_newPauseTimestamp);
      emit Paused(_newPauseTimestamp);
  }
```

**Note:**

> It should also be noted that this bug also impacts the `VaultUpgradeableBeacon` contract of currently live Kiln Defi `v1.0` protocol.

**Kiln:** Fixed in PR 231.

**Spearbit:** `ConnectorRegistry` has been fixed. For what `VaultUpgradeableBeacon` concerns, as the beacon contract is immutable, the bug will be fixed by granting `PAUSER_ROLE` to a `PauserProxy` contract which performs the overflow check.

### 5.1.2 ConnectorRegistry: CONNECTOR_MANAGER_ROLE can unpause a connector without having UNPAUSER_ROLE

**Severity:** Medium Risk

**Context:** ConnectorRegistry.sol#L215-L218

**Description:** The ConnectorRegistry.remove function is missing the check to validate the paused state of the connector. Due to this the CONNECTOR_MANAGER_ROLE can unpause a paused connector without having the UNPAUSER_ROLE.

**Scenario:**

1. A connector gets added to ConnectorRegistry by CONNECTOR_MANAGER.
2. After a while the PAUSER account pauses the connector.
3. Now the CONNECTOR_MANAGER can call ConnectorRegistry.remove function to delete the connector state (including the paused state) and can then call ConnectorRegistry.add which will re-register the connector with a fresh unpaused state.

**Proof of Concept:**

```
function test_poc_ConnectorRegistry_managerCanUnpause() public {
    address manager = makeAddr("manager");
    address account = makeAddr("account");
    ConnectorRegistry registry = new ConnectorRegistry(account, account, account, account, manager, 0);
    address connector = address(new ConnectorMock());
    bytes32 name = bytes32("connector-1");

    vm.prank(manager);
    registry.add(name, connector);
    vm.prank(account);
    registry.pause(name);
    assertEq(registry.paused(name), true);
    vm.startPrank(manager);
    registry.remove(name);
    registry.add(name, connector);
    assertEq(registry.paused(name), false);
    vm.stopPrank();
}
```

**Recommendation:** Consider validating the paused state of connector in remove function.

```
    function remove(bytes32 name) external override exists(name) whenNotFrozen(name)
↪   onlyRole(CONNECTOR_MANAGER_ROLE) {
+       if (paused(name)) revert ConnectorPaused(name);
        delete connectorInfo[name];
        emit ConnectorRemoved(name);
    }
```

**Kiln:** Fixed in PR 231.

**Spearbit:** Verified.

## 5.2 Low Risk

### 5.2.1 `ConnectorRegistry.paused` returns `false` value for non-existent connector names

**Severity:** Low Risk

**Context:** ConnectorRegistry.sol#L184-L186

**Description:** The `ConnectorRegistry.paused` function does not validate the existence of the input `name` parameter. Due to this it returns a valid `false` value for a non-existent connector name.

**Recommendation:** Consider adding the `exists` modifier:

```diff
- function paused(bytes32 name) public view override returns (bool) {
+ function paused(bytes32 name) public view override exists(name) returns (bool) {
      return connectorInfo[name].pauseTimestamp > block.timestamp;
  }
```

**Kiln:** Fixed in PR 231.

**Spearbit:** Verified.

### 5.2.2 The `Vault` implementation cannot receive native chain token as reward

**Severity:** Low Risk

**Context:** Vault.sol#L948

**Description:** The `Vault` contract implements the `claimAdditionalRewards` function to handle all possible rewards provided by an external protocol. These additional rewards can be swapped and/or sent as per the project's requirement.

However the `Vault` contract lacks the ability to receive native chain tokens as additional rewards. So in case an external protocol starts rewarding its users in native chain tokens (like ETH) then Kiln vaults will not be able to receive and handle those rewards.

Considering the possibility of integrating with new protocols on different chains, support for native token rewards could be a necessary feature for vaults.

**Recommendation:** Consider adding a `receive` function in `Vault` to receive and handle native chain tokens.

**Kiln:** Acknowledged. After working on the fix, the receive function is crushing the bytecode size limit. We might use some workarounds or add the function later if needed.

**Spearbit:** Acknowledged.

### 5.2.3 `FeeDispatcher.dispatchFees` is susceptible to re-entrancy attack

**Severity:** Low Risk

**Context:** FeeDispatcher.sol#L95-L134

**Description:** The `FeeDispatcher.dispatchFees` function exposes a re-entrancy attack vector as it updated the internal storage states after performing the external call.

```
function dispatchFees(IERC20 asset, uint8 underlyingDecimals) external {

    // ...

    for (uint256 i; i < _recipientsLength; i++) {
        currentRecipient = $._dispatches[msg.sender]._feeRecipients[i];

        if (_pendingDepositFee > 0) {
            // ...
            if (_depositFeeAmount > 0) {
                asset.safeTransferFrom(msg.sender, currentRecipient.recipient, _depositFeeAmount);
↪   // @audit reentrancy
                _depositFeeTransferred += _depositFeeAmount;
                // ...
            }
        }
        if (_pendingRewardFee > 0) {
            // ...
            if (_rewardFeeAmount > 0) {
                asset.safeTransferFrom(msg.sender, currentRecipient.recipient, _rewardFeeAmount);
↪   // @audit reentrancy
                _rewardFeeTransferred += _rewardFeeAmount;
                // ...
            }
        }
        // ...
    }
    $._dispatches[msg.sender]._pendingDepositFee = _pendingDepositFee - _depositFeeTransferred;
    $._dispatches[msg.sender]._pendingRewardFee = _pendingRewardFee - _rewardFeeTransferred;
}
```

Technically during the `asset.safeTransferFrom` call it is possible to re-enter and call the `dispatchFees` function again which will result in a mismatch between the token distributed and `$._dispatches` state update.

Currently to exploit this for `Vault`, the `asset` needs to be a token with transfer hooks (ERC777) and recipient should have a way to force a Vault to call `FeeDispatcher.dispatchFees` in between an existing call. As `Vault.dispatchFees` has a `nonReentrant` modifier vaults are not exposed to this re-entrancy attack. But there is no protection in `FeeDispatcher` itself against re-entrancy.

`FeeDispatcher` is a standalone contract which is accessible to everyone. If any other contract interacts with `FeeDispatcher` then that contract will be exposed to re-entrancy attack. Offloading the re-entrancy protection to the other contract (like Vault) is not a good design.

**Recommendation:** Consider adding the `nonReentrant` modifier to `FeeDispatcher.dispatchFees` function to eliminate all re-entrancy risks.

**Kiln:** Fixed in PR 231.

**Spearbit:** Verified.

### 5.2.4 Vault upgrade process does not reset the old `FeeDispatcherStorage` state of vault

**Severity:** Low Risk

**Context:** Vault.sol#L405-L413

**Description:** During a vault upgrade, the `Vault.__Vault_upgrade()` function initializes all new vault storage states that are necessary for new `Vault` implementation. It sets the new `FeeDispatcher.Dispatch` states by calling `incrementPendingDepositFee`, `incrementPendingRewardFee` & `setFeeRecipients` functions on new external `FeeDispatcher` contract.

However it does not reset the old `FeeDispatcherStorage` internal states of the vault. These old and outdated states of vault remain non-zero forever. While these old does not currently impact the vault operations they can impact future vault implementations and upgrades.

**Recommendation:** Consider explicitly resetting the old `FeeDispatcherStorage` internal state of the vault to `null` values during the vault upgrade.

**Proof of Concept:** Add this test in `test/migration/migration.t.sol`:

```
function test_poc_migrationLeavesOldState() external {
    _singleArchiveDeposit(alice, 100 ether);
    _checksAndMigrate();

    bytes32 FeeDispatcherOldStorageLocation =
↪   0xfdd5e928c3467d3da929a44639dde8d54e0576a04fec4ff333caa67a6f243300;
    bytes32 old_pendingManagementFee = address(vault).readStorage(FeeDispatcherOldStorageLocation);
    expect(uint256(old_pendingManagementFee)).toEqual(1 ether);
}
```

**Kiln:** Acknowledged. After looking for at a fix. It seems that the solution is not straightforward. We need to reset the storage in the `__getFeeDispatcherStorage` function (used for the `delegatecall`). But since we are returning the storage, we must directly return the values (cached in memory) after resetting the storage. We can't do this inside the `__Vault_upgrade` function, since we removed the informations linked to the `FeeDispatcher`.

**Spearbit:** Acknowledged. Note that since the old `FeeDispatcher_1_0_0.FeeDispatcherStorage` and new `IFeeDispatcher.Dispatch` structs are exactly similar to each other (only names changed), a solution like this works:

```
  function __Vault_upgrade(UpgradeParams calldata params) internal onlyInitializing {
      _setBlockList(params.blockList_);
      _setAdditionalRewardsStrategy(params.additionalRewardsStrategy_);
      _setFeeDispatcher(params.feeDispatcher_);
      IFeeDispatcher(params.feeDispatcher_).incrementPendingDepositFee(params.pendingDepositFee_);
      IFeeDispatcher(params.feeDispatcher_).incrementPendingRewardFee(params.pendingRewardFee_);
      IFeeDispatcher(params.feeDispatcher_).setFeeRecipients(params.recipients_, _underlyingDecimals());
      SafeERC20.forceApprove(IERC20(asset()), params.feeDispatcher_, type(uint256).max);

+     bytes32 FeeDispatcherStorageLocation =
↪   0xfdd5e928c3467d3da929a44639dde8d54e0576a04fec4ff333caa67a6f243300;
+     IFeeDispatcher.Dispatch storage $;
+     assembly {
+         $.slot := FeeDispatcherStorageLocation
+     }
+     delete $._pendingDepositFee;
+     delete $._pendingRewardFee;
+     delete $._feeRecipients;
  }
```

Note that resetting of old `_feeRecipients` has not been tested, but `_pendingDepositFee` and `_pendingRewardFee` are definitely getting reset using this method. Please do create tests to make sure that `_feeRecipients` are also getting reset.

### 5.2.5 Blind transfers in `Vault.forceWithdraw` can cause loss of funds in specific scenarios

**Severity:** Low Risk

**Context:** Vault.sol#L1010

**Description:** If a smart contract is `isBlockedByInternalList` and does not have code to handle blind transfers (e.g. when using internal accounting instead of `balanceOf(address(this))`, calling `Vault.forceWithdraw` will lead to loss of funds.

**Recommendation:** Prefer pull over push for fund transfers.

**Kiln:** Acknowledged. We can assume that the `Blocklist` operator is aware of the situation by blocking an integration contract.

**Spearbit:** Acknowledged.

### 5.2.6 Inaccurate event emission on calling `forceWithdraw`

**Severity:** Low Risk

**Context:** Vault.sol#L1027

**Description:** The `Withdraw` event (see Vault.sol#L1027) event emitted shows the `sanctionedUser` as sender when they are not who made the call.

**Recommendation:** The simple modification of passing in `msg.sender` would revert however as the allowance check occurs when `caller != owner` in _withdraw'.

An alternative is to force approve prior to calling the internal `_withdraw` function or to modify the `_withdraw` function itself (the latter is not recommended).

**Kiln:** Acknowledged.

**Speabit:** Acknowledged.

### 5.2.7 `Vault.maxDepoist` and `Vault.maxMint` do not conform to ERC4626 spec

**Severity:** Low Risk

**Context:** Vault.sol#L686

**Description:** The `Vault` and several connectors perform an equality check against `type(uint256).max - 1` when there is no limit on deposit amounts. Per the spec `MUST return 2 ** 256 - 1 if there is no limit on the maximum amount of assets that may be deposited.`

The implementation issue is that `2 ** 256 - 1` != `type(uint256).max - 1` and the functions should be using `type(uint256).max` instead.

Some of the connectors are fine as the protocol they integrate with are correct, they simply expend more gas arriving at the correct return value. Other connectors, such as Venus for example, return a value that is 1 less than the expected value:

```
function test_MaxDeposit() public {
    vault.maxDeposit(alice);
    expect(vault.maxDeposit(alice)).toEqual(type(uint256).max);
}
```

**Recommendation:** Replace `type(uint256).max - 1` with `type(uint256).max` in the Vaults and connectors. CAUTION: this could be a breaking change if those integrating with Kiln expect the off by 1 value to be present.

**Kiln:** Fixed in commit 2b7896bb.

**Spearbit:** Verified.

### 5.2.8 `Vault.forceWithdraw`: sanctioned users can withdraw their assets in case they are in the internal block list

**Severity:** Low Risk

**Context:** Vault.sol#L1013

**Description:** The system maintains two different block lists, the OFAC block list (sanctions list) which is based on a contract based on Chainalysis, and an internal list integrators can manage. `forceWithdraw` allows anyone to send the funds of internal block listed users back to them but is currently also allows OFAC sanctioned users to do that as well, while it should not, according to the spec.

**Recommendation:** Consider reverting the call to `forceWithdraw` in case the user (`sanctionedUser`) is part of the OFAC sanction list.

**Kiln:** Fixed in commit e3716da2.

**Spearbit:** Verified.

## 5.3 Gas Optimization

### 5.3.1 `FeeDispatcher.setFeeRecipients`: nested loops can be replaced with a single loop based on a sorted array

**Severity:** Gas Optimization

**Context:** FeeDispatcher.sol#L218

**Description:** `setFeeRecipients` is deduplicating the parameter of `recipients` by using nested arrays. This can be optimized by using only one loop that enforces strictly ascending order for this array, then the array should be submitted sorted (off-chain).

**Kiln:** Acknowledged. We prefer to keep things on-chain. It was also raised during the first audit.

**Spearbit:** Acknowledged.

### 5.3.2 Gas Optimizaton

**Severity:** Gas Optimization

**Context:** SUSDSConnector.sol#L60

**Description:**

1. SUsds ignores the param in the same way the connector does (see SUSDSConnector.sol#L29). Can save gas by passing in the 0 address (null bytes are cheaper).
2. `BlockListFactory.CreateBlockListParams` is identical to `BlockList.InitializationParams` and could be substituted for it saving the duplication in memory (see BlockListFactory.sol#L90-L96).
3. Using calldata is cheaper than memory in `createBlockList` (BlockListFactory.sol#L85),`addToBlockList` (BlockList.sol#L130), and `removeFromBlockList` (BlockList.sol#L140).

**Kiln:** We acknowledge the first optimization because the sUSDS is upgradeable and the second for clarity even if they are identical. The third optimization has been applied in commit 35f3837d.

**Spearbit:** Verified.

## 5.4 Informational

### 5.4.1 `multisend`: Consider adding the recipient address that reverted the call to the error message

**Severity:** Informational

**Context:** MultisendLib.sol#L32

**Description:** `multisend` has made a design decision to revert the entire call in case one of the calls to `IERC20(token).safeTransfer(...)` reverts. token transfers to recipients might revert in case one recipient is blacklisted for example, or in case of a deliberate revert in ERC777 tokens. It is important to mention that this described scenario will not result in a permanent denial of service since the caller (`CLAIM_MANAGER_ROLE`) can specify the recipients array.

**Recommendation:** Consider wrapping the external call in a try and catch block and revert the call with the recipient address for better visibility in case the call reverts.

**Kiln:** Acknowledged. We think that the natspec is misleading (also forgot about it). We are sending only one token at multiple recipients, and not multiple tokens. In the end, the way it reverts is fine. We ended up editing the natspec for more clarity in commit 2b1490ba.

**Spearbit:** Acknowledged.

### 5.4.2 Missing checks to prevent out of bound access of storage arrays

**Severity:** Informational

**Context:** FeeDispatcher.sol#L171-L174, VaultFactory.sol#L290-L292

**Description:** The `FeeDispatcher.feeRecipientAt` and `VaultFactory.getDeployedVault` functions directly reads the input `index` value of storage arrays without validating that `index < array.length`. This leads an EVM `Panic` exception (`panic: array out-of-bounds access (0x32)`).

**Recommendation:** Consider adding an explicit check to validate that the input `index` is always less than the `array.length`.

**Kiln:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.4.3 External functions of connectors can be accessed directly

**Severity:** Informational

**Context:** AaveV3Connector.sol#L115-L118, IConnector.sol#L22-L26

**Description:** The external functions of connectors are intended to be `delegatecall`ed by the vaults. But as per the current implementation of connector functions they can be directly `call`ed by anyone.

**Recommendation:** Consider implementing an `onlyDelegateCall` modifier in connectors and add it to all `external` functions that the `IConnector` interface exposes.

**Kiln:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.4.4 All `view` functions of vault will start reverting when vault's beacon is paused

**Severity:** Informational

**Context:** VaultUpgradeableBeacon.sol#L80-L83, VaultUpgradeableBeacon.sol#L123-L125

**Description:** The `VaultUpgradeableBeacon.implementation` function contains the `whenNotPaused` modifier. This modifier reverts the call when beacon contract is paused. In case the vault beacon contract gets paused then all non state changing functions (`view` & `pure`) of vault will always revert. This behaviour can make reading storage states of paused vaults very difficult.

**Recommendation:** Beacon will only be paused in critical situations and reading vault states will be a crucial need in those situations. A workaround for this issue is to add a generic storage reader function in `VaultBeaconProxy` (not the beacon). Something like this:

```
function extsload(bytes32 slot) external view returns (bytes32) {
    assembly ("memory-safe") {
        mstore(0, sload(slot))
        return(0, 0x20)
    }
}
```

You can also implement an advanced version which starts reading the storage from `VaultStorageLocation` and reads the entire `VaultStorage` struct as a `bytes32[]` array. Note that reading storage state like this is already possible via the `eth_getStorageAt` rpc method.

**Kiln:** Acknowledged (and the `VaultBeaconProxy` is immutable). In case of paused contracts we will use a specific script/simulation that can make all view calls. Even if it's not useful for the users, it's only for us.

**Spearbit:** Acknowledged.

### 5.4.5 Use of Openzeppelin library version with a known bug in `Base64.encode` function

**Severity:** Informational

**Context:** .gitmodules#L7-L12

**Description:** The protocol uses openzeppelin library version `5.0.0` which contains a bug in the `Base64.encode` function as described in report `GHSA-9vx6-7xxf-x967`.

Note that the current protocol contracts do not use the impacted `Base64.encode` function.

**Recommendation:** It is recommended to upgrade the openzeppelin library to the patched `5.0.2` version.

**Kiln:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.4.6 Typos and comments

**Severity:** Informational

**Context:** VaultUpgradeableBeacon.sol#L172, Vault.sol#L151

**Description/Recommendation:**

1. `VaultUpgradeableBeacon.sol#L172`: Change comment to 'Can only be called by the current unpauser'.
2. `Vault.sol#L151`: Missing natspec for `_additionalRewardsStrategy` parameter.

**Kiln:** Fixed in PR 231.

**Spearbit:** Verified.

### 5.4.7 Code style and quality notes

**Severity:** Informational

**Context:** BlockListFactory.sol#L99

**Description/Recommendation:**

1. The blockList address (BlockListFactory.sol#L99) does not need to be payable.
2. Checking `== true` (BlockList.sol#L162) is unnecessary as it's comparing against a boolean. Can instead return `return ISanctionsList($._underlyingSanctionsList).isSanctioned(addr) || $._blockList[addr];`
3. `forceWithdraw` (Vault.sol#L1010) internal mechanics resemble `redeem` rather than `withdraw` and would be better renamed to `forceRedeemAll` or `forceRedeem`.
4. `forceWithdraw` (Vault.sol#L1010) refers to `sanctionedUser` but checks only the internal block list. These users would be more accurately described as `blockedUser`.

**Kiln:** Fixed 1, 2 and 4 in commit 868a3ba0. 3 is acknowledged, even if we are redeeming. It's more straightforward to say `forceWithdraw` since we offer only one way of doing it.

**Spearbit:** Verified.

### 5.4.8 Blocked and sanctioned accounts may still receive funds

**Severity:** Informational

**Context:** Vault.sol#L561-L562

**Description:** `notBlocked` checks `sender` and `owner` so sanctioned and blocked accounts cannot withdraw or make the call withdraw. `receiver` is still permitted to receive the withdrawal even if sanctioned.

**Recommendation:** Consider adding blocking recipient as well.

**Kiln:** Acknowledged. We considered it unnecessary to check if the receiver is blocked when depositing and withdrawing.

- When depositing, the blocked receiver won't be able to withdraw the funds and could be forced to withdraw (with forceWithdraw).
- When withdrawing, even if a blocked receiver can't receive the assets, you can withdraw and send the assets afterward. This is an easy workaround. Checking that the receiver is blocked seems "*out-of-scope*".

**Spearbit:** Acknowledged.

### 5.4.9 Additional Reward dust Accumulates in the contract

**Severity:** Informational

**Context:** MultisendLib.sol#L51

**Description:** Due to rounding down, dust rewards will accumulate in the contract when using multisend.

**Recommendation:** Track total transferred and for the final index in the array send `total - totalTransferred` and validate the amount is greater than or equal to `total.mulDiv(_split, _scaledMaxPercent)`.

**Kiln:** Acknowledged.

**Spearbit:** Acknowledged.

### 5.4.10 Missing `_disableInitializers` call for Vault implementation

**Severity:** Informational

**Context:** VaultFactory.sol#L244, Vault.sol#L401

**Description:** The `Vault` contract lacks a way to prevent initialization of vault implementation contract. So technically it is possible to call `upgrade` function on the Vault implementation contract.

The `DEPLOYER_ROLE` account on `VaultFactory` can pass the vault implementation contract address to `VaultFactory.upgradeVault` function due to which the `upgrade` function will get executed on the vault implementation contract.

**Recommendation:** Consider adding this in `Vault` contract:

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

**Kiln:** Acknowledged. The call to `upgrade()` will fail with an `InvalidInitialization()` error since you need to initialize the `Vault` to be able to upgrade. Also, the `VaultFactory` can only initialize new vault proxies and not the implementation. For this issue to exist, we need a function in the `VaultFactory` that can initialize a given address.

We removed the `onlyDelegateCall` modifier from the proxy functions (which prevented this) because the introduced `onlyFactory` modifier with the current implementation also prevents the issue. Even if adding the `_disableInitializers()` is not a big deal, I prefer to keep the modification scope smaller.

**Spearbit:** Acknowledged.

### 5.4.11 Missing check to prevent initialization of FeeDispatcher implementation contract

**Severity:** Informational

**Context:** *(No context files were provided by the reviewer)*

**Description:** The changes made in PR 231 adds an `initialize` function in `FeeDispatcher`.

```
function initialize() public {
    _initialize();
}

function _initialize() internal initializer {
    __ReentrancyGuard_init();
}
```

However no check exists to prevent calling of `initialize` function on FeeDispatcher implementation contract.

**Recommendation:** Consider adding an `onlyDelegateCall` modifier to `FeeDispatcher.initialize` similar to other upgradable contracts in the protocol.

**Kiln:** Fixed in commit 08bf46a4.

**Spearbit:** Verified.