Kiln DeFi Integrations Security Review

Auditors

Gerard Persoon, Lead Security Researcher

R0bert, Lead Security Researcher

Report prepared by: Lucas Goiriz

May 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 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 10 days in total, Kiln engaged with Spearbit to review the defi-integrations protocol. In this period of time a total of **36** issues were found.

Summary

Project Name	Kiln
Repository	defi-integrations
Commit	aa938dc5
Type of Project	Vaults, Staking
Audit Timeline	May 7th to May 17th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	6	0	6
Low Risk	13	0	13
Gas Optimizations	4	0	4
Informational	13	0	13
Total	**36**	**0**	**36**

The present report's objective is to check any remaining risks to deploy immutable Kiln DeFi vaults on top of immutable Morpho vaults. All findings are acknowledged because the identified risks will be managed by surrounding code and surrounding procedures.

5 Findings

5.1 Medium Risk

5.1.1 Function `delegateToFactory()` gives too much power to the factory

Severity: Medium Risk

Context: Vault.sol#L423-L425

Description: Because the `VaultFactory` can be upgraded, it can call the function `delegateToFactory()` on all the deployed Vaults. Function `delegateToFactory()` does do a delegatecall back to the `VaultFactory`. This way the `VaultFactory` can do everything to the Vaults including draining all the funds.

For comparison: `VaultUpgradeableBeacon` and the`ConnectorRegistry` have a freeze function to remove upgrades risks. However for the `VaultFactory` there is no freeze function.

Recommendation: To make the contracts immutable, the `VaultFactory` has to be locked down in a way it can't be upgraded. This can be done via `renounceOwnership()` of the ProxyAdmin that manages the EIP-1967 Transparent Proxy of the VaultFactory. Alternatively the function `delegateToFactory()` could be removed.

Kiln: Acknowledged, we will see when it's possible to freeze the factory.

Spearbit: Acknowledged.

5.1.2 Early `claimAdditionalRewards` call can inflate shares and DoS future deposits

Severity: Medium Risk

Context: Vault.sol#L979-L990

Description: Within the `Vault` contract, if the `CLAIM_MANAGER_ROLE` calls `claimAdditionalRewards` prematurely, before the `Vault` has any meaningful liquidity, subsequent large deposits can fail with a `MinimumTotalSupplyNotReached()` error. This is because the assets of the `Vault` are increased by the rewarded amount but the `totalSupply` would remain 0. Therefore, when then, a future user calls `deposit`, the asset to share ratio is totally skewed and the user is forced to provide an amount of assets considerably higher than the initially rewarded amount to be able to pass the following check present in the `_deposit` function:

```
if (totalSupply() < $._minTotalSupply) revert MinimumTotalSupplyNotReached();
```

Because the `Vault` is designed to maintain a minimum supply of shares after each deposit, having a reward claim event first can put the `Vault` in an unexpected state and could be abused by a user with the `CLAIM_MANAGER_ROLE` to put the `Vault` into a Denial of Service state.

Proof of Concept: Fuzz run logs:

```
NewState(1)
vaultDispatchFees;Timestamp(1746621455)
vaultClaimAdditionalRewards;RewardsClaimed(453459583721357243528 3);Timestamp(1746960078)
NewState(2)
ERR_vaultDeposit;User(0xeeec67a5dfa5f8cf1ce446343d9b62116efc190b);AssetAmount(430693002626156734101891 9
↪  69025);Error(0xb086de13);Timestamp(1746960078) // <---
INVARIANT_FAILED;InvariantId(_invariant_AssetsToSharesToAssets);InitialAssets(3101582851934741975356);S 
↪  hares(0);AssetsFinal(0)
```

Console logs:

```
      [25980] 0x7f39C581F595B53c5cb19bD0b3f8dA6c935E2Ca0::transferFrom(0xeeEc67a5DfA5F8Cf1ce446343D9b6
↪  2116EfC190B, 0x804749760fE7e479148bF5c593aCACF4BDcbd556, 430693002626156734101891969025 [4.306e29])
         emit Transfer(from: 0xeeEc67a5DfA5F8Cf1ce446343D9b62116EfC190B, to:
↪  0x804749760fE7e479148bF5c593aCACF4BDcbd556, value: 430693002626156734101891969025 [4.306e29])
         emit Approval(owner: 0xeeEc67a5DfA5F8Cf1ce446343D9b62116EfC190B, spender:
↪  0x804749760fE7e479148bF5c593aCACF4BDcbd556, value: 0)
         ← [Return] true
      emit Transfer(from: 0x0000000000000000000000000000000000000000, to:
↪  0xeeEc67a5DfA5F8Cf1ce446343D9b62116EfC190B, value: 94979358 [9.497e7])
      ← [Revert] MinimumTotalSupplyNotReached()
   ← [Revert] MinimumTotalSupplyNotReached()
```

Notice the huge deposit of `430693002626_156734101891969025` wstETH. This deposit only converts to `94979358` shares:

```
emit Transfer(from: 0x0000000000000000000000000000000000000000, to:
↪  0xeeEc67a5DfA5F8Cf1ce446343D9b62116EfC190B, value: 94979358 [9.497e7])
```

Recommendation: First decide on the finding "Multiple incoherent mitigations for the inflation attacks". If `_minTotalSupply` will be used then do the following:

- Enforce a check preventing `claimAdditionalRewards` from being called when the vault's total supply is below a certain threshold, ensuring the first major activity is not a reward claim.

Kiln: Acknowledged. We can check `if (totalSupply() < $._minTotalSupply)` inside the `claimAdditionalRewards` function.

Spearbit: Acknowledged.

5.1.3 Small amounts of assets can be withdrawn without paying shares

Severity: Medium Risk

Context: Vault.sol#L559-L580

Description: In function `withdraw()`, the call to `_roundDownPartialShares()` rounds down. Due to this potentially `assets` can be withdrawn without having to pay `shares` for it.

> Note: currently `_decimalsOffset()` is 0 in the deploy scripts and all deployments, which means `_roundDownPartialShares()` effectively does nothing and there is currently no risk.
> Note: If `_decimalsOffset()` is not 0, the round down is usually small. With an asset like `BTC` and 8 decimals the amount of assets gained by the round down would be `$100_000 / 1e8 ~ $ 0.001`.

However on chains with low gas costs this could be profitable. Also this is a reputational risk.

Recommendation: First decide on the finding "Multiple incoherent mitigations for the inflation attacks". Change the code to do a `roundUp`, similar to the call to `_convertToShares()`, which uses `Math.Rounding.Ceil`. Alternatively make sure `_decimalsOffset()` is always 0.

Kiln: Acknowledged. For now only offset of 0 will be used.

Spearbit: Acknowledged.

5.1.4 Multiple incoherent mitigations for inflation attacks

Severity: Medium Risk

Context: Vault.sol#L633

Description: There are multiple solutions used to prevent inflation attacks.

1. Virtual shares, which are implemented in `convertToShares()` and `convertToAssets()` in both ERC4626Upgradeable and Vault.
2. `_minTotalSupply` in Vault.

However they are not used optimally:

- A high value of `_decimalsOffset()` reduces the risk, however in practice a value of 0 is always used;
- `_roundDownPartialShares()` and `_checkPartialShares()` are added to reduce the negative impact of `_decimalsOffset()`, however isn't really used because `_decimalsOffset()` is 0. Additionally a vulnerability has been introduced in the situation where `_decimalsOffset()` is larger than 0. See finding Small amounts of assets can be withdrawn without paying shares.
- `_minTotalSupply()` can lead to a denial of service (DOS) if shares of the underlying vault are deposted first. See findings Early claimAdditionalRewards call can inflate shares and DoS future deposits and Unrestricted direct deposit into underlying metamorpho vault can inflate main vault's totalAssets and trigger a DoS;
- `_minTotalSupply()` is only enforced on `_deposit()` and not on other function that `_mint()` and `_burn()` shares, like `_withdraw`, `collectRewardFees()` and `_accrueRewardFee()`.

Recommendation: Consider choosing one of the approaches:

1. Use `_decimalsOffset() == 0`. In that case all `PartialShares` logic could be removed as well as the `_minTotalSupply` logic. This gives a basic protection against the inflation attack.
2. Use `_decimalsOffset() > 0`. However this isn't practical if the underlying asset already has 18 decimals. Make sure `PartialShares` logic is used consistently and safe. Also the `_minTotalSupply` logic can be removed. This gives a good protection against the inflation attack.
3. Use `_minTotalSupply` logic. In that case also add it to all functions that `_mint()` and `_burn()` shares, like `_withdraw`, `collectRewardFees()` and `_accrueRewardFee()`. Then the `PartialShares` and `_decimalsOffset()` could be removed in Vault. Also the basic `convertToShares()` and `convertToAssets()` from `ERC4626Upgradeable` should be overridden. This gives a good protection against the inflation attack.

Even in combination with these technical solutions it is important to deposit funds right after deployment and check this wasn't frontrun. Also see Morpho risks.

Kiln: Acknowledged. We will consider to make a fix to `_minTotalSupply` logic (recommendation 3). We will keep the offset (to keep the patch small).

Spearbit: Acknowledged.

5.1.5 `claimAdditionalRewards-reinvest` calls can be sandwiched

Severity: Medium Risk

Context: Vault.sol#L979

Description: Because calling `Vault.claimAdditionalRewards` will inject more assets into the `Vault` without simultaneously increasing the share supply, a malicious actor could easily sandwich the call for a profit:

1. Front-run deposit: The attacker deposits a large amount of the underlying asset right before `claimAdditionalRewards` is called. At this moment, shares still cost the old "price," not factoring in the imminent rewards injection.
2. Rewards pulled into the Vault as new assets: `claimAdditionalRewards` executes and increases the `Vault`'s total assets by reinvesting the extra tokens. Consequently, each share is effectively backed by more assets than before.
3. Back-run withdraw: Immediately after the rewards are reinvested into the `Vault`, the attacker withdraws (redeems) the shares, capturing a disproportionate portion of the newly claimed rewards. They end up withdrawing more assets than they would have if the claim hadn't happened, essentially siphoning away rewards that should belong to longer-term participants.

On the other hand, there is also another possible second attack vector which involves a user with the `CLAIM_MANAGER_ROLE` performing a flashloan:

1. Flashloan deposit: The claim manager takes a flashloan of the vault's asset, deposits it into the `Vault`, then calls `claimAdditionalRewards`.
2. Withdraw: The claim manager redeems or withdraws right after the claim, returning the flashloan plus trivial fees while pocketing nearly all the newly claimed rewards. This effectively diverts rewards to the attacker who used the flashloan.

Recommendation: The only way to fully prevent the sandwich attack around the claim of the additional rewards is enforcing a mandatory lock period on deposits which is not really the best solution. Therefore, consider documenting the possibility of sandwich attacks so integrators and end users understand that, if no lock is in place, skilled MEV participants or role holders could siphon off newly claimed rewards.

Kiln: Acknowledged. We will not implement a patch. This will be managed on the operational side.

Spearbit: Acknowledged.

5.1.6 Unrestricted direct deposit into underlying metamorpho vault can inflate main vault's totalAssets and trigger a DoS

Severity: Medium Risk

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

Description: In Metamorpho vaults, any user can directly call the `deposit(assets, receiver)` function, specifying the main `Vault` as the receiver. By doing so, shares are minted directly to the main `Vault` increasing its asset balance:

```
// msg.sender = main vault
function totalAssets(IERC20) external view returns (uint256) {
    return metamorpho.previewRedeem(metamorpho.balanceOf(msg.sender));
}
```

Crucially, this "external deposit" occurs without a corresponding share mint in the `Vault`, causing an assets–shares mismatch from the perspective of the `Vault`'s own accounting. If the `Vault` is empty or near-empty, or if `_minTotalSupply` is set, having externally inflated `totalAssets()` can cause a scenario where the next actual deposit reverts with `MinimumTotalSupplyNotReached()`, denying any deposit into the `Vault`.

This vector is particularly evident for newly deployed Vaults with no user liquidity. But, because the `Vault`'s address is deterministically derived by the factory through the use of `CREATE2`, an attacker can deposit underlying tokens directly in the Metamorpho underlying vault before or right after the Main Vault is deployed, pumping instantly its `totalAssets()` and once again pushing it into a DoS state.

Proof of Concept: Here is a description what happens with an initial donation:

```
Deploy a vault with 6 decimals, _minTotalSupply = 1e6 and _decimalsOffset = 0
Deposit 1e6 assets in the underlying vault, with receiver == deployed Vault (e.g. an initial donation of
↪  1e6)

1) Deposit 1e6 assets in the deployed Vault. Now you would receive 0 shares but don't meet
↪  _minTotalSupply.
2) Start again and deposit 1000001 assets in the deployed Vault. Now you would receive 1 shares but
↪  don't meet _minTotalSupply.
3) Start again and deposit 1e12+1e6 assets in the deployed Vault. Now `_minTotalSupply` is reached and
↪  you receive 1e6 shares.
```

Here is some solidity code that does these calculations:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.29;
import "hardhat/console.sol";
import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

contract inflate {
    using Math for uint256;
    uint _minTotalSupply = 1e6;
    uint totalSupply;
    uint totalAssets;
    uint _decimalsOffset=0;
    function _convertToShares(uint256 assets ) internal view returns (uint256) {
        return assets.mulDiv(totalSupply + 10 ** _decimalsOffset, totalAssets + 1);
      }
    function previewDeposit(uint256 assets) public view returns (uint256) {
        return _convertToShares(assets);
    }
    constructor() {
        totalAssets = 1e6; // initial donation
        console.log(previewDeposit(1e6)); //0
        console.log(previewDeposit(1e6+1)); //1
        console.log(previewDeposit(1e12+1e6)); //1000000
    }
}
```

Recommendation: Upon deploying the Kiln Vault, require that an official deposit be made to set a valid baseline. This ensures the `Vault`'s initial `totalAssets()` and share supply are in a consistent state, preventing any external party from inflating the asset count first.

On the other hand, document that anyone can inflate the `Vault`' s asset by depositing underlying tokens directly in the Metamorpho underlying vault before the main `Vault` was even deployed. If that's the case, consider reverting the deployment so the deployer can re-try the deployment through the `VaultFactory` with a different `salt`.

Kiln: Acknowledged. But will see if we can include a check inside the initialization (that `totalAssets() == 0`).

Spearbit: Acknowledged.

5.2 Low Risk

5.2.1 No check asset underlying vault is same as asset of Vault

Severity: Low Risk

Context: ConnectorRegistry.sol#L197, ConnectorRegistry.sol#L206, Vault.sol#L364-L369, Vault.sol#L407-L415, Vault.sol#L1106-L1119

Description: There is no explicit check that the asset of the underlying vault is the same as the asset of the Vault. Due to this configuration mistakes might not be quickly detected.

Recommendation: Consider checking the assets are the same, during deployment `__Vault_init()` and upgrade `__Vault_upgrade()` of the Vault and during a change of the connector contract in `update()`. Function `_setConnectorRegistry()` and `_setConnectorName()` should also check this, if they would be exposed to an external function.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.2 Updating a `connector` introduces risks

Severity: Low Risk

Context: ConnectorRegistry.sol#L206-L216, ConnectorRegistry.sol#L259

Description: Changing the connector could have different reasons:

1. To fix something in the connector, while the underlying vault stays the same (new swaptarget, new logic).
2. To replace the underlying vault due to an issue with the underlying vault (outdated, bug, hack, rugpull).

In case of 2 additional risk occur:

- `totalAssets()` will initially be 0 and a small deposit will result in large amount of shares;
- The old shares of the underlying vault are still in the vault.

Note: the `freeze()` function in `FeeDispatcher` prevents updates once its activated.

Recommendation: In case of 1 it would be good to confirm the underlying vault indeed stays the same. In case of 2:

Make a plan for this scenario, which could include the following:

- make sure the replaced underlying vault has as much value as feasible and add the shares for that to the Vault;
- Cconsider moving the shares from the previous underlying vault out of the contract to be able to salvage funds;
- Have a way to later add any salvaged funds, without it being seen as reward.

Kiln: Interesting, I had never considered replacing the underlying protocol, as it essentially requires a migration. We don't provide any security guarantees beyond the integration itself, so it's a scenario that's hard to envision except for correcting the implementation. Acknowledged.

Spearbit: Acknowledged.

5.2.3 Not all init functions are called

Severity: Low Risk

Context: BlockList.sol#L113-L116, ExternalAccessControl.sol#L61-L64, VaultFactory.sol#L117-L118, Vault.sol#L350-L362

Description: The `initialize()` should call all init functions of the underlying libraries. However some init functions are not called. The risk is limited because these functions currently are empty, but this could change in the future, although these functions are empty. Same for `BlockList` and `VaultFactory`.

Recommendation: Consider adding the following init functions:

- `__AccessControl_init()`.
- `__ERC165_init()`.
- `__Context_Init()`.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.4 Functions `FeeDispatcher` require consistent input

Severity: Low Risk

Context: FeeDispatcher.sol#L129

Description: The functions in `FeeDispatcher` only work correctly if the correct `asset` and `underlyingDecimals` are supplied. For example if `dispatchFees()` is called with the shares of the underlying vault then these could be transferred out. `Connector`s have the possibility to call `FeeDispatcher` as if it is the vault, due to the `delegatecall`

and could abuse the function of `FeeDispatcher`. The calculation of `_pendingDepositFee` and `_pendingRewardFee` only works if its always the same `asset` and the same `underlyingDecimals`.

Recommendation: The connectors are trusted and should be checked carefully to not allow interactions with the `FeeDispatcher`. In `dispatchFees()` consider using `IERC4626(msg.sender).asset()` and `IERC20Metadata(asset).decimals()`.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.5 Overly restrictive checks for `deposit()` and `mint()`

Severity: Low Risk

Context: Vault.sol#L507, Vault.sol#L518-L519, Vault.sol#L532, Vault.sol#L547-L548

Description: When calling the `Vault.deposit` function, the vault checks if `assets > _maxDeposit()` and reverts if so. However, a portion of assets is taken as a deposit fee and never goes to the underlying protocol, so the check should only compare the net (after-fee) assets being deposited to `_maxDeposit()`. Similarly, in the `Vault.mint` function, the vault checks `if (shares > _maxMint())`, but it does not subtract the portion of shares lost to deposit fees, leading to a stricter-than-necessary limit on how many shares users can mint.

Recommendation: Adjust these checks to account for the portion that is paid as a deposit fee before it goes to the underlying protocol. Specifically, use the net deposit/asset values (or net minted shares) in the `_maxDeposit` / `_maxMint` comparisons.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.6 Rounding in `_previewDeposit()` and `_previewMint()` allows users to bypass the deposit fee

Severity: Low Risk

Context: Vault.sol#L736-L738, Vault.sol#L783-L784

Description: In the `Vault._previewDeposit` and `Vault._previewMint` functions, the deposit fee is calculated as:

```
// _previewDeposit
depositFeeAmount = assets.mulDiv($._depositFee, _MAX_PERCENT * 10 ** _underlyingDecimals());

// _previewMint
depositFeeAmount = assets.mulDiv(_depositFee, _MAX_PERCENT * 10 ** _decimals, Math.Rounding.Floor);
```

which always round down. This can be exploited by splitting a large deposit into many small deposits. Because each small deposit's fee is rounded down, the total paid over multiple small deposits can be less than the fee assessed by making a single large deposit. As a result, the "pending deposit fee" may be under-collected compared to the intended rate.

Recommendation: Consider always rounding up in the deposit fee calculations.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.7 Check for `partialshares` not always done

Severity: Low Risk

Context: Vault.sol#L437-L470, Vault.sol#L480-L483, Vault.sol#L498-L504

Description: The functions `previewMint()` , `previewRedeem()`, `maxMint()`, `maxRedeem()`, `maxWithdraw()`, `maxDeposit()` don't use/check `partialshares`. On the other hand function `mint()`, `redeem()`, `withdraw()`,

`deposit ()`, `previewDeposit()` and `previewWithdraw()` do use/check `partialshares`. This might lead to inconsistent result for callers of these functions.

Recommendation: First decide on the finding "Multiple incoherent mitigations for the inflation attacks". If `partialshares` will be used then doublecheck if the current implementation is desirable and consider changing it.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.8 Additional rewards can not be claimed if the underlying `metamorpho` vault has reached its maximum cap

Severity: Low Risk

Context: Vault.sol#L979-L982

Description: The `Vault.claimAdditionalRewards` function can revert if the `AdditionalRewardsStrategy` is set to `Reinvest` and the underlying Metamorpho vault's maximum deposit cap has already been reached (or is exceeded by the amount being claimed and re-deposited). This would trigger a revert with an `AllCapsReached()` error (`0xded0652d`). Consequently, the vault would not be able to reinvest the additional rewards. Users who expected yield from those extra rewards are blocked from realizing that yield until some existing liquidity exits the metamorpho vault, freeing up capacity. This state could persist indefinitely if the vault remains near its maximum deposit cap.

Recommendation:

- For vault operators, document that reinvest calls can fail if the underlying protocol has reached its deposit cap and consider letting the claim manager switch strategies to `Claim` instead of `Reinvest` in such scenarios.
- Optionally, integrate a check on the vault side that detects if the underlying deposit would exceed the Metamorpho cap and handles it gracefully (e.g., partial reinvest or a fallback path).
- In user-facing documentation, clarify that once the underlying protocol's deposit cap is reached, further reinvest attempts may revert, so additional rewards will remain uninvested until enough liquidity is withdrawn.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.9 Function `claimAdditionalRewards()` doesn't restrict `rewardsAsset`

Severity: Low Risk

Context: MetamorphoConnector.sol#L71-L73, Vault.sol#L952-L967

Description: Function `claimAdditionalRewards()` doesn't restrict the used `rewardsAsset`, however `claim()` does restrict it: it disallows `addres(this)` and `address(metamorpho)`, which are the two types of `shares` being used. However there is no restriction on the use of the `asset()`. A relatively limited amount of `asset()` can be in the Vault after `collectRewardFees()`. The claim manager can thus use `claimAdditionalRewards()` to retrieve `asset()` from the Vault. This is a trusted actor so this normally shouldn't happen.

Recommendation: As it is safer to enforce restrictions in the contracts consider to add in function `claimAdditionalRewards()` a revert if `rewardsAsset == asset()`.

Note: then it is also important that `asset()` isn't a dual address token like Monerium EURe, see Monerium V1 and V2.

Also consider moving the check for `address(rewardsAsset) == address(this)` inside of function `claimAdditionalRewards()` because this is relevant for all connectors.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.10 Function `setRewardFee()` call be called even when nonReentrant flag is set

Severity: Low Risk

Context: Vault.sol#L1074-L1078

Description: Function `setRewardFee()` can be called while another function of Vault is executing and the `nonReentrant` flag is set, because it doesn't have a `nonReentrant` modifier. As `setRewardFee()` calls `_accrueRewardFee()` it could potentially use a temporarily balance, which it then uses to mint shares. For example `claimAdditionalRewards()` could do an external call via `reinvest()`, which could call `setRewardFee()`.

Note: this would require trusted operators like the feeManager and the claimManager to be malicious and colluding and possibly requires a compromised connector so this is very unlikely in practice.

Recommendation: To be safe consider adding a `nonReentrant` modifier to `setRewardFee()`.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.11 `upgradeTo()` could do more checks

Severity: Low Risk

Context: VaultUpgradeableBeacon.sol#L132-L134, VaultFactory.sol#L117-L124

Description: `VaultFactory::initialize()` checks that `vaultFactory()` of the vault implementation matches the factory. However in `VaultUpgradeableBeacon`, when the implementation is upgraded via `upgradeTo()` this check isn't done. This is inconsistent.

Recommendation: Consider checking `Vault(newImplementation).vaultFactory() == vaultFactory` in `upgradeTo()`. *Note: currently `VaultUpgradeableBeacon` doesn't know the value of `vaultFactory`*.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.12 `multisend()` has no duplicate check

Severity: Low Risk

Context: FeeDispatcher.sol#L222, FeeDispatcher.sol#L252-L256, MultisendLib.sol#L32

Description: Function `FeeDispatcher::incrementPendingRewardFee()` has a check for duplicate recipients. However the comparable function `multisend()` doesn't check this, which is inconsistent.

Recommendation: Consider adding a duplicate check.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.2.13 The function `decimals()` is not mandatory for ERC20 tokens

Severity: Low Risk

Context: MultisendLib.sol#L32-L36

Description: The function `decimals()` is not mandatory for erc20 tokens to this could in theory be missing for the (reward)Token.

Recommendation: If you want to handle this situation, consider using `_tryGetAssetDecimals()` of OZ.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.3 Gas Optimization

5.3.1 Gas optimizations

Severity: Gas Optimization

Context: FeeDispatcher.sol#L146, Vault.sol#L624-L648, Vault.sol#L656-L676, Vault.sol#L771-L784, Vault.sol#L833-L835

Description: Some gas optimizations can be done.

Recommendation:

- `FeeDispatcher::dispatchFees()` could cache `_MAX_PERCENT * 10 ** underlyingDecimals`.
- `Vault::_deposit()` could cache `asset()`.
- `Vault::_withdraw()` could cache `asset()`.
- `Vault::_accruedRewardFeeShares()` could cache `$._rewardFee`.
- `Vault::_previewMint` could cache `_MAX_PERCENT * 10 ** underlyingDecimals`.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.3.2 Transfers in `dispatchFees()` can be combined

Severity: Gas Optimization

Context: FeeDispatcher.sol#L148, FeeDispatcher.sol#L160

Description: In function `dispatchFees()` there are currently two transfers to the same `recipient`. Combining them will save some gas.

Recommendation: Consider combining the transfers to one transfer by summing `_depositFeeAmount` and `_rewardFeeAmount`.

Kiln: Acknowledged, we preferred the idea of separating for indexing (transfer then event for each).

Spearbit: Acknowledged.

5.3.3 Transient storage can be used for the `ReentrancyGuard`

Severity: Gas Optimization

Context: Vault.sol#L358

Description: The library ReentrancyGuardUpgradeable is used. There is also a version that uses transient storage that is more efficient.

Recommendation: Consider using `ReentrancyGuardTransientUpgradeable`. Then check the chain on which the contract are deployed support transient storage.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.3.4 Function `_underlyingDecimals()` retrieves `decimals()` every time

Severity: Gas Optimization

Context: Vault.sol#L1266-L1268

Description: Function `_underlyingDecimals()` retrieves `decimals()` every time the function is called. This requires a contract call, which is relatively expensive.

Recommendation: Consider retrieving `decimals()` it once and store it.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4 Informational

5.4.1 The name `getOrRevert()` is not descriptive

Severity: Informational

Context: ConnectorRegistry.sol#L173-L180, IConnectorRegistry.sol#L20-L23

Description: The name `getOrRevert()` is not very descriptive: function `get()` also reverts when the name isn't found. The difference is that `getOrRevert()` additionally also checks for paused.

Recommendation: Consider changing the name to something like `getWithPauseCheck()`.

Kiln: Acknowledged, the behaviour is documented in `IConnectorRegistry`.

Spearbit: Acknowledged.

5.4.2 Public `convertToShares` / `convertToAssets` vault functions should never be used directly by integrators

Severity: Informational

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

Description: The Vault's public `convertToShares` and `convertToAssets` do not include any unaccrued reward fees. As a result, integrators who rely on these functions to see how many shares they'll receive or how many assets they'll redeem may get inaccurate results if some portion of yield has accumulated but not yet minted as fee shares.

Recommendation: Document clearly that these conversions only reflect the vault's current minted supply and do not auto-accrue or factor in pending reward fees. Where accurate, updated conversions are required, integrators should rely on the different "preview" functions that incorporate new yield.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.3 Functions `previewWithdraw()` and `previewRedeem()` have similar code to other functions

Severity: Informational

Context: Vault.sol#L487-L495, Vault.sol#L498-L504, Vault.sol#L789-L805

Description: Functions `previewWithdraw()` and `previewRedeem()` have code that is very similar to `_convertToShares()` and `_convertToAssets()`. This code duplication makes the source more difficult to understand and maintain.

Recommendation: Consider using `_convertToShares()` in `previewWithdraw()` and `_convertToAssets()` in `previewRedeem()`.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.4 Nested function calls with side effects are difficult to verify

Severity: Informational

Context: Vault.sol#L574

Description: Function `withdraw()` calls both `_accrueRewardFee()` and `totalSupply()` while calling `_convertToShares()`. As `_accrueRewardFee()` changes `totalSupply()`, which is used in the next argument, the result depends on the implementation of the compiler. The current code works correctly but takes effort to understand, verify and maintain.

Recommendation: Consider calling `_accrueRewardFee()` in the beginning of the function, similar to the way `redeem()` does it.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.5 Return parameters not always documented

Severity: Informational

Context: BlockListFactory.sol#L115, BlockList.sol#L52, Vault.sol#L157, Vault.sol#L422-L423, Vault.sol#L679-L680, Vault.sol#L686-L687, Vault.sol#L698-L699, Vault.sol#L706-L707, Vault.sol#L789, Vault.sol#L798-L799, Vault.sol#L1014-L1015, Vault.sol#L1254-L1255

Description: For several functions the return parameters are not documented.

Recommendation: Consider also documenting the return parameters.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.6 `_accruedRewardFeeShares()` called inconsistently

Severity: Informational

Context: Vault.sol#L498-L504, Vault.sol#L699-L701, Vault.sol#L707-L721

Description: Function `_maxWithdraw()` calls `previewRedeem()`, which calls `_accruedRewardFeeShares()`. However the comparable `_maxRedeem()` doesn't call `_accruedRewardFeeShares()`. This doesn't seem logical.

Recommendation: Consider calling `_accruedRewardFeeShares()` consistently.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.7 Different ways to round down

Severity: Informational

Context: Vault.sol#L738, Vault.sol#L784

Description: The functions `_previewMint()` and `_previewDeposit()` use a different way to round down. One uses the implicit version and the other the explicit version.

Recommendation: Consider using the same approach in both functions.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.8 Checks with dual `checkTransferability()` might be too restrictive

Severity: Informational

Context: Vault.sol#L248-L251, Vault.sol#L281-L291, Vault.sol#L559-L567, Vault.sol#L583-L591, Vault.sol#L870-L880, Vault.sol#L883-L895

Description: The functions `transferFrom()`, `redeem()` and `withdraw()` have two `checkTransferability()` restrictions, which are `AND`ed together. For function `transferFrom()` these are for both `from` and `to`. This means when `msg.sender` doesn't have the `SPENDER_ROLE`, then both `from` and `to` must have the `SPENDER_ROLE` to be able to do `transferFrom()`.

Note: `from` must already have given an allowance to the `msg.sender` to enable `transferFrom()`.

If `from` would initiate the `transfer()`, it would be allowed to transfer towards a `to` that doesn't have the `SPENDER_ROLE`. A similar situation is present with `redeem()` and `withdraw()`. So the checks with dual `checkTransferability()` might be too restrictive.

Recommendation: Double check if the behaviour is as intended and change the code if it is too restrictive.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.9 Similar code in `collectRewardFees()` and `collectableRewardFees()`

Severity: Informational

Context: Vault.sol#L920-L930, Vault.sol#L1209-L1214

Description: Functions `collectRewardFees()` has code that is very similar to `collectableRewardFees()`. This code duplication makes the source more difficult to understand and maintain.

Recommendation: Consider calling `collectableRewardFees()` from `collectRewardFees()`:

```
  function collectRewardFees() external nonReentrant onlyRole(FEE_COLLECTOR_ROLE) {.
      // ...
-     (uint256 _rewardFeeShares, uint256 _newTotalAssets) = _accruedRewardFeeShares();
-     uint256 _collectable = _convertToAssets( ... ).
+     uint256 _collectable = collectableRewardFees();
      // ...
  }
```

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.10 `forceWithdraw()` can give better error messages

Severity: Informational

Context: Vault.sol#L1015-L1025

Description: In function `forceWithdraw()`, if `_blockList` isn't set (yet), then the function will revert without a specific error message. This might make troubleshooting more difficult.

Recommendation: Consider verifying that `_blockList` is set and revert with a specific error message if it isn't.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.11 Inherent risks of `rewardsAsset` and `swapTarget` call

Severity: Informational

Context: MetamorphoConnector.sol#L71-L88, MetamorphoConnector.sol#L91-L117, MultisendLib.sol#L32-L54

Description: The functions `claim()`, `reinvest()` and `multisend()` call function of a supplied `rewardsAsset`. This could in theory be any contract and could allow for reentrant calls. The function `claim()`, calls `claimInfo.distributor.claim()`, which could in theory be any contract and could allow for reentrant calls. The function `reinvest()` calls a supplied function of `swapTarget` with supplied parameters. This could send out all `rewardsAsset` and also do reentrant calls. These are all inherent risks of the current approach.

Recommendation: Make sure the `claimManager` is fully trusted. If any of the risks above are unexpected, add additional checks in the code.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.12 `pauseFor()` workaround

Severity: Informational

Context: ConnectorRegistry.sol#L238-L249, PauserProxy.sol#L59-L64, VaultUpgradeableBeacon.sol#L160-L170

Description: `VaultUpgradeableBeacon::pauseFor()` should also use `toUint88()`, like `ConnectorRegistry::pauseFor()` does, to prevent truncating the timestamp. This is added for completeness. The project is aware of this and has created the work around of `PauserProxy`.

Recommendation: Once upgrades are done on the contracts also apply this fix.

Kiln: Acknowledged.

Spearbit: Acknowledged.

5.4.13 `multisend()` can leave dust

Severity: Informational

Context: MultisendLib.sol#L32-L53

Description: The function `multisend()` can leave some dust of `token` behind due to rounding errors. This will be minimal so can also be ignored.

Recommendation: If you want to prevent leaving dust, in the last iteration of the loop, the remaining amount could be sent to the recipient.

Kiln: Acknowledged.

Spearbit: Acknowledged.

6 Appendix

6.1 Fuzzing Suite Methodology and Invariants

6.1.1 Methodology

Spearbit was tasked with developing a fuzzing suite, which was designed by r0bert, to be run against five different production-deployed Kiln DeFi Vault contracts. This suite aims to simulate real-world interactions and verify the system's correctness under diverse conditions. The methodology follows a systematic and structured approach to uncover logical inconsistencies, edge cases and potential vulnerabilities:

1. **State Machine Design:**
   - The suite operates as a state machine, executing a sequence of random actions (e.g., deposits, withdrawals) across multiple states.
2. **Mainnet Forking:**
   - Testing occurs on an Ethereum mainnet fork at block 22431806 (May 7, 2025), replicating production-like conditions for realistic results.
3. **Vault and User Simulation:**
   - A vault is randomly selected from a predefined set of deployed vaults (e.g., Safe Smokehouse wstETH) or a mock vault is deployed with a specified asset (e.g., USDC).
   - Pseudo-random users are created with varying balances of the vault's underlying asset, simulating diverse user profiles and behaviors.
4. **Action Probabilities:**
   - Actions are assigned dynamic probabilities. Early states prioritize deposits (e.g., 30% chance) to initialize the vault, while later states balance operations like withdrawals and reward claims.
5. **Invariant Verification:**
   - After each action, a predefined set of invariants is checked to ensure the contract maintains its expected properties and state consistency.
6. **Error Handling and Logging:**
   - Expected errors (e.g., `NothingToCollect()` triggered when there are no rewards available yet) are whitelisted and logged without failing the test, while unexpected errors cause a revert and are flagged for review.

This methodology ensures a comprehensive coverage of the vault's functionality, stress-testing its resilience and correctness across a wide range of realistic scenarios.

6.1.2 Invariants Implemented

The fuzzing suite enforces a robust set of invariants to validate the vault's behavior and integrity after each action. These invariants focus on accounting accuracy, share consistency and operational correctness. They are detailed in the table below:

Invariant	Description
`_invariant_TotalSupplyIsSumOfBalances`	Ensures `totalSupply == sum(user balances) + collectable reward fees.`
`_invariant_AssetsMoreThanSupply`	Verifies `totalAssets * 10^offset >= totalSupply` (with rounding tolerance).
`_invariant_TotalAssetsTotalSupplyConsistency`	Checks `convertToAssets(totalSupply) <= totalAssets.`

Invariant	Description
`_invariant_AssetsToSharesToAssets`	Ensures converting assets to shares and back approximates the original assets (±10).
`_invariant_ViewFunctionsDoNotRevert`	Confirms view functions (e.g., `totalAssets`, `maxDeposit`) never revert.
`invariant_TotalAssetsAlwaysIncreaseAfterADeposit`	Assets must increase after a deposit or mint operation.
`invariant_TotalSharesAlwaysIncreaseAfterADeposit`	Shares must increase after a deposit or mint operation.
`invariant_TotalAssetsAlwaysDecreaseAfterAWithdrawal`	Assets must decrease after a withdrawal or redeem operation.
`invariant_TotalSharesAlwaysDecreaseAfterAWithdrawal`	Shares must decrease after a withdrawal or redeem operation (adjusted for reward fees).
`invariant_AssetsDecreaseAfterAWithdrawalMatchUserReceived`	The decrease in assets after a withdrawal must match the assets received by the user.
`invariant_PendingDepositFeeIncreasesAfterDepositOrMint`	Pending deposit fee must increase after a deposit or mint when `depositFee > 0`.
`invariant_MainVaultAccountingIsCorrect`	Preview calculations must match actual results for deposit, mint, withdraw, and redeem operations (±0.1).
`invariant_UserSharesMultipleOfOffset`	Ensures each user's share balance is a multiple of `10^offset`.

These invariants collectively safeguard the vault's core functionality, ensuring that asset and share balances align with expected outcomes, user interactions are processed correctly and the system remains stable under varied conditions.