PUBLIC

# Code Assessment

# of the Multivault Smart Contracts

February 11, 2025

Produced for

Mellow

by

CHAINSECURITY

# Contents

# 1 Executive Summary

Dear all,

Thank you for trusting us to help Mellow Finance with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Multivault according to Scope to support you in forming an opinion on their security risks.

Mellow Finance implements an upgrade to the previously audited vaults to support multiple LRTs at once as well as other ERC-4626 compliant protocols. The multivault architecture implements a modular integration framework by leveraging adapters for integrations and strategies for allocations. Given the delayed withdrawals for LRTs, specialized withdrawal queues have been implemented.

The most critical subjects covered in our audit are asset solvency, functional correctness, and frontrunning. The general subjects covered are upgradeability, unit testing, documentation, and trustworthiness. Note that testing is insufficient and that some of the uncovered issues could've been caught by testing.

The most notable findings are:

- The Incorrect Valuation of ERC-4626 that leads to incorrect share prices potentially leading to a loss of funds.
- An Escalation of Privileges that could allow addresses with low privileges to drain the protocol.
- Architectural problems such as Pending Assets Become Claimed During Withdrawal and Insufficient Limitations for Strategies that could have lead to DoS scenarios.
- Integration issues such as Operator Undelegations Are Not Accounted for that could've led to loss of funds.

All issues have been resolved through code corrections or specification change. Some lower severity issues have been acknowledged, and their risk has been accepted.

Additionally, please consult Notes, Assessment Overview and Trust Model and Roles for considerations that could be out of scope.

In summary, we find that the codebase provides a good but improvable level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 2 |
| • Code Corrected | 2 |
| Medium-Severity Findings | 7 |
| • Code Corrected | 5 |
| • Specification Changed | 1 |
| • Risk Accepted | 1 |
| Low-Severity Findings | 19 |
| • Code Corrected | 13 |
| • Code Partially Corrected | 1 |
| • Risk Accepted | 2 |
| • Acknowledged | 3 |

# 2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

## 2.1 Scope

The assessment was performed on the source code files inside the Multivault repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 2 Dec 2024 | 764c15a91f6c322c30b9d4a736aa9d31ea06e3c7 | Initial Version |
| 2 | 13 Dec 2024 | 74ce3cc8feddaf862744095969b52852b49c0920 | Version with fixes |
| 3 | 19 Dec 2024 | 733df8430261584a7bfb53fcd5ce9d11b4e7123a | After intermediate report |
| 4 | 15 Jan 2025 | d11e531e24cfa08a6d9c3ca92031468f3a4ceb71 | After intermediate report |
| 5 | 20 Jan 2025 | 3b38213c942b2ed2ac5e6b28b69b77cd5592995e | After intermediate report |
| 6 | 20 Jan 2025 | ca6b05bc665878c9aa059a3df7c8df6ec5194093 | Final Version |
| 7 | 04 Feb 2025 | 528b9ecd2aa016fe24aea98e15b14f941ef56cb9 | Migrator |
| 8 | 10 Feb 2025 | 65ce8dde861678f5d00bd3ac5bd458a4bf523f7f | Final Migrator |

For the solidity smart contracts, the compiler version `0.8.25` was chosen.

As of version 1, the contracts included in scope were:

```
src/
    adapters/
        ERC4626Adapter.sol
        SymbioticAdapter.sol
    queues/
        SymbioticWithdrawalQueue.sol
    strategies/
        RatiosStrategy.sol
    utils/
        Claimer.sol
    vaults/
        MultiVault.sol
        MultiVaultStorage.sol
```

In version 3, the following contract were included in scope:

```
src/
    adapters/
        EigenLayerAdapter.sol
        EigenLayerWstETHAdapter.sol
        IsolatedEigenLayerVault.sol
        IsolatedEigenLayerVaultFactory.sol
        IsolatedEigenLayerWstETHVault.sol
        IsolatedEigenLayerWstETHVaultFactory.sol
    utils/WhitelistedEthWrapper.sol
    queues/
        EigenLayerWithdrawalQueue.sol
```

In version 4, the following contracts were included in scope:

```
src/
    queues/
        EigenLayerWstETHWithdrawalQueue.sol
```

In version 4, the following contracts were removed from scope:

```
src/
    adapters/
        IsolatedEigenLayerWstETHVaultFactory.sol
```

In version 5, the following contracts were removed from scope:

```
src/
    queues/
        EigenLayerWstETHWithdrawalQueue.sol
```

In version 7, the following contracts were added to the scope:

```
src/utils/
    Migrator.sol
    EigenLayerFactoryHelper.sol
```

### 2.1.1 Excluded from scope

All other contracts were excluded from scope.

Note that the following contracts were out of scope as they were part of a previous review:

```
src/
    utils/
        EthWrapper.sol
    vaults/
        ERC4626Vault.sol
        VaultControl.sol
        VaultControlStorage.sol
```

Note that the following contracts are only partially in scope as the previous reviews included that contract in scope (only differences are in scope):

```
src/queues/SymbioticWithdrawalQueue.sol
```

Note that their usage in the audited context is not out of scope.

Any other integration should be carefully reviewed in the context of the metavault. This review cannot account for all potential external protocols and integrations. All external dependencies and contracts are out of scope and are expected to work as documented.

Only very standard ERC-20 tokens are supported by the system. If external systems support certain tokens, that does not imply the Mellow Finance protocol supports that.

The review is limited to the outlined trust model, see Trust Model and Roles.

## 2.2 System Overview

This system overview describes the latest version received (Version 6) of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Mellow Finance implements ERC4626-like vaults that allow allocating the respective underlying token to several external protocols at once (typically LRTs). The integration-specific logic is delegated to specialized adapters while the allocation is defined in smart contracts outlining strategies.

### *2.2.1 Multi Vault*

The `MultiVault` is the central contract for a given underlying token. It tokenizes deposits, allocates funds to subvaults and allows configuring the multivault.

**ERC-4626.** Deposits are tokenized as shares of a tokenized vault. While the multivault generally matches the ERC-4626 standard, there are notable deviations (see ERC-4626 violations for more details). Note that the base ERC-4626 `ERC4626Vault` (part of a previous audit), on which the multivault is built upon, adds the following features on top of the ERC-4626 functionality:

- Deposit limit: A soft limit on deposits can be set with `VaultControl.setLimit()`.
- Pausing: Deposits and withdrawals can be paused with `VaultControl.pauseDeposits()` / `VaultControl.unpauseDeposits()` and `VaultControl.pauseWithdrawals()` / `VaultControl.unpauseWithdrawals()`, respectively. Note that a pauser loses their respective role on pausing actions.
- Recipient whitelist: : A recipient whitelist can be set with `VaultControl.setDepositWhitelist()`. In case the whitelist is active a user can not receive any deposit. Addresses can be (un-)whitelisted with `VaultControl.setDepositorWhitelistStatus()`.

Note that the settings are accounted for in regard to the ERC-4626 functionality.

**Subvaults and External Protocols.** The multivault can deposit into multiple external protocols at once. Hence, the multivault has a set of subvaults (external protocol contracts) that correspond to the respective external protocol's vault. The protocols supported currently are:

- Symbiotic: Underlying funds can be deposited into Symbiotic vaults.
- EigenLayer: Underlying funds can be deposited into EigenLayer vaults. Strategies restaking ERC20 tokens are supported (restaking Validator ETH is not).

- ERC-4626: While ERC-4626 is not a protocol, all protocols implementing standard ERC-4626 vaults can be supported.

To support integrating with one of the above protocols, the corresponding adapter can be registered with the respective functions `MultiVault.setSymbioticAdapter()`, `MultiVault.setEigenLayerAdapter()` and `MultiVault.setERC4626Adapter()` by privileged addresses. Once a protocol is supported, privileged addresses can configure a subvault with `MultiVault.addSubvault()` and `MultiVault.removeSubvault()`.

When a subvault is registered, the corresponding adapter can create a withdrawal queue to allow handling delayed withdrawals that are typically part of LRT architectures. Note that if no queue is needed, no queue needs to be created. For more details regarding withdrawal queues, see Withdrawal Queues.

The adapter implements integration-specific logic such as depositing, withdrawing, reward-claiming as well as getters for retrieving relevant data (e.g. valuation protocol shares, allocation limits). For more details, see Adapters.

**Fund Allocation.** The multivault's managed funds will be held in various forms representing the underlying asset. Hence, the funds will be distributed across subvault and can be buffered locally as liquid balances. The funds will be (de-)allocated automatically during deposits and withdrawals while privileged addresses can trigger rebalancing with `MultiVault.rebalance()` to reallocate funds. All allocations are defined in strategies which can be set with `MultiVault.setDepositStrategy()`, `MultiVault.setWithdrawalStrategy()` and `MultiVault.setRebalanceStrategy()`, respectively. For more details regarding strategies, see Allocation Strategies.

The funds can be held in the forms below:

- Liquid ERC-20 balance: Unallocated amounts are held in form of underlying ERC-20 token.
- Liquid Symbiotic Default Collateral balance: As much as possible from liquid ERC-20 balance will be deposited into Symbiotic's default collateral. The default collateral can be set at most once with the function `MultiVault.setDefaultCollateral()`. In case it is not set, all the liquid balance is held as a liquid ERC-20 balance.
- Subvault balance: Funds deposited into external protocols will be held externally. However, some shares of the external vault will be typically received in return (e.g. ERC-4626 token).
- Pending queue balance: Due to rebalancing, the multivault could be awaiting withdrawals from withdrawal queues.
- Claimable queue balance: The pending balances will eventually become claimable. Hence, the multivault could hold claimable balances in the withdrawal queues.

The valuation function `ERC4626.totalAssets()` will thus sum the pending and claimable queue balances as well as the subvault balances for all subvaults with the liquid balances.

Below, the allocation mechanisms are outlined in more detail:

- Deposits: Before the funds are pulled from the user, the deposit strategy is queried for the deposit amounts for the given asset delta. Once the funds are received, the funds are deposited according to the strategy's plan. Unallocated funds will be kept as liquid balances as outlined above. Note that `Adapter.maxDeposit()` will return the maximum deposit for a given subvault.
- Withdrawals: By querying the withdrawal strategy for withdrawal amounts, a plan for withdrawals is created, according to which the withdrawals are performed. Note that for each subvault/queue pair, that includes protocol withdrawals (potentially delayed in form of queue shares) as well as claimable amounts and pending amounts in form of queue shares. Any remainder is taken from the liquid balances.
- Rebalancing: By querying the rebalancing strategy for amounts to claim, withdraw and deposit, a rebalancing plan is crafted. First, all claiming and protocol withdrawals are performed. Then, all deposits are performed. Unallocated funds will be kept as liquid balances as outlined above. Note that rebalancing can lead to withdrawal requests owned by the multivault which leads to the possibility of pending and claimable balances in withdrawal queues.

**Rewards.** Privileged roles can add support for claiming rewards from a protocol with `MultiVault.setRewardsData()`. Note that the following is defined for each registered farm:

- Curator treasury and curator fee: The curator fee is the share that a curator will receive to the curator treasury.
- Distribution farm: The distribution farm is the destination where the remaining rewards will be sent to (note: no checkpointing mechanism is implemented, and the farm is not notified).
- Protocol specific data: That includes the reward token, the protocol (required for the adapter) and some data needed for protocol specific rewards claiming (e.g. farm contract).

The rewards can be pushed to the distribution farm and the curator treasury with `MultiVault.pushRewards()`.

Note that farms can be removed by specifying the reward token in `MultiVault.setRewardsData()` to be `0x0`.

### *2.2.2 Adapters*

Adapters define integration-specific logic.

**General.** The multivault requires that adapters satisfy certain properties. Below is a list of functionalities along with the expected properties.

- `vault()`: Must return the vault interacting with the adapter. Only the vault returned shall be allowed to use the state-modifying functions.
- `assetOf()`: Must return the underlying token of a given suitable subvault as the function is used for checking a subvault's token compatibility.
- `handleVault()`: Must return a suitable withdrawal queue or `0x0` if no withdrawal queue is needed. Should return the same withdrawal queue for a subvault if it is readded.
- `stakedAt()`: Must return the value held in a given subvault by the multivault (not including queue-related amounts).
- `maxDeposit()`: Must return the maximum depositable amount so that `deposit()` can succeed.
- `deposit()`: Must deposit the given amount into the given subvault. Must handle approval logic. Does not consider any protocol-specific limitations. Used with delegatecalls during deposits and rebalances.
- `withdraw()`: Must withdraw the given amount from the given subvault. Note that this may include withdrawal queue related logic. Does not consider any protocol-specific limitations. Used with delegatecalls during withdrawals and rebalances.
- `validateRewardData()`: Must validate the data provided when registering a farm as much as possible.
- `pushRewards()`: Must claim rewards to the vault accordingly. Note that it must support decoding the registered data stored by the multivault. Similarly, the data provided by users must be validated if necessary. Used with delegatecalls when pushing rewards.
- `areWithdrawalsPaused()`: A function to identify whether withdrawals are paused.

Additionally, none of the functions should revert with data considered valid (e.g. an amount lower than the maximum deposit is deposited should not revert).

**Symbiotic Adapter.** The Symbiotic adapter will create a `SymbioticWithdrawalQueue`.

**ERC-4626 Adapter.** The ERC-4626 adapter will not create a withdrawal queue as withdrawals are instantaneous for regular ERC-4626 tokens. The functionality related to rewards will revert given that rewards are not part of the standard. Withdrawals are treated as paused if `ERC4626.maxRedeem()` returns zero if the vault has a non-zero balance of the token.

**EigenLayer Adapter.** The EigenLayer adapter interacts with EigenLayer through a separate contract, `IsolatedEigenLayerVault`, for each whitelisted EigenLayer strategy. The isolated vault is permissionlessly deployed together with `EigenEigenLayerWithdrawalQueue` from the factory contract `IsolatedEigenLayerVaultFactory`. For the wstETH strategy, the contracts `IsolatedEigenLayerWstETHVault` and `IsolatedEigenLayerWstETHVaultFactory` are used instead, since the EigenLayer strategy uses stETH. Therefore, wstETH is unwrapped before being deposited and rewrapped after being withdrawn from the strategy. Adapter functions `sharesToUnderlyingView` and `underlyingToShares` are used to convert between EigenLayer shares and assets. If EigenLayer uses a different asset than the MultiVault (e.g., wstETH), the adapter must implement the conversion from the EigenLayer asset to the underlying asset.

### 2.2.3 Allocation Strategies

Allocation strategies define how the allocation of funds to subvaults is managed.

**General.** The multivault requires that allocation strategies satisfy certain properties. Below is a list of functions defined in the deposit, withdrawal and rebalancing strategy, respectively, along with expected properties.

- `calculateDepositAmounts()`: Computes the amounts to deposit to each subvault for a given asset delta to deposit. Cannot exceed the asset delta. Must respect any limitations given by subvaults. That includes the `MultiVault.maxDeposit()` for a given subvault as well as any other limitations.
- `calculateWithdrawalAmounts()`: Computes the withdrawal amounts from subvaults for an asset delta to withdraw which includes a withdrawn amount from the protocol, a claimable amount from the queue and a pending amount from the queue. Each amount must be lower than the respective value queried from adapters and queues. The sum of all amounts cannot exceed the asset delta. The remainder must be coverable by liquid balances. Any other limitations must be respected.
- `calculateRebalanceAmounts()`: Computes the amounts to claim, the amounts to withdraw and the amounts to deposit. Must respect all the properties above. However, note that no asset deltas are defined. However, the sum of deposits cannot exceed the sum of the liquid balances and the claimable amounts.

**Ratio Strategy.** The `RatioStrategy` contract serves as a deposit, withdrawal and rebalancing strategy that can be shared among multivaults. It implements a strategy that defines a minimum and maximum share of a vault's TVL for each subvault that can be set for a given vault with `RatioStrategy.setRatio()` by privileged roles. All operations aim to keep the allocation of a subvault in the respective ranges. Below is a more detailed description of the mechanism.

- Deposit: Allocates the deposit delta in the following order of preference: fill subvault deficits, fill subvault maxima, liquid balances.
- Withdraw: Fills the withdrawal delta in the following order of preference: liquid balances, excess subvault balances violating the maxima, subvault balances over the minima, pending assets, claimable assets, remaining subvault balances.
- Rebalance: Withdraws any subvault excess balances. Aims to fill all subvault deficits. In case the deficits cannot be covered by the liquid and the claimable balances (immediately depositable) as well as the pending and subvault excess balances (will eventually become depositable), withdraws from subvaults, respecting their minima to try to fill the remainder. Deposits the coverage for deficits that can be immediately deposited. In case of a remainder of immediately usable balances, the remainder is deposited to all subvaults respecting their respective maxima.

*Note that withdrawals and rebalancing will be blocked if one of the underlying subvaults withdrawals are locked (e.g. pausing).* See Adapters for more details.

### 2.2.4 Withdrawal Queues

The withdrawal queues manage delayed withdrawals for users and for multivaults.

**General.** Similar to the allocation strategies and adapters, the multivault requires that withdrawal queues satisfy certain properties. Below is a list of expected functions along with the expected behaviour.

- `pendingAssetsOf()`: Returns the amount of assets pending for a given address.
- `claimableAssetsOf()`: Returns the amount of assets claimable for a given address.
- `claim()`: Claims assets for an address up to a maximum amount. Expected that for a user, only the user and the `Claimer` can claim on behalf of the user.
- `transferPendingAssets()`: Transfers pending assets registered under a user's name to another user. In the case of accounting with shares (or similar), this must round down.
- `pull()`: Expected to pull all claimable funds from the external protocol.
- `claimer()`: Getter for the privileged claimer.

Note that the adapters might require more protocol- and queue-specific functionality.

**Symbiotic Withdrawal Queue.** The `SymbioticWithdrawalQueue` manages withdrawal requests for one multivault and one Symbiotic vault. It implements a share-based mechanism per epoch to fairly distribute the requests made through the queue. The requests are created by multivaults through adapters calling `SymbioticWithdrawalQueue.request()`, creating requests owned by the queue. Note that all functions typically handle pending epochs for a given user. However, pending epochs can be additionally handled with `SymbioticWithdrawalQueue.handlePendingEpochs()` for a given user. Note that other functions are provided as additional getters. Note that only the correct usage of the queue and the implementation of function `transferPendingAssets()` have been in scope. The queue has been in scope of a prior audit.

**EigenLayer Withdrawal Queue.** The `EigenLayerWithdrawalQueue` manages withdrawal requests for each EigenLayer strategy. Similarily, the `EigenLayerWstETHWithdrawalQueue` withdrawals for the stETH strategy only. The queue behaves similarly to symbiotic queues but with some key differences: EigenLayer withdrawals are not epoch-based; instead, each withdrawal request is individually timestamped and can be claimed after a certain delay (the maximum of global and strategy-specific delays). Each withdrawal request from EigenLayer creates a new entry in the withdrawal queue that is split via a share-based mechanism. To avoid out-of-gas issues, the number of withdrawal request is limited to 14. Pending assets sent to other users are not directly accounted towards them; instead, beneficiaries must call the function `acceptPendingAssets()` to accept pending assets. Any contract that expects pending assets must implement a function to accept pending assets. Similar to the Symbiotic queue, the function `handleWithdrawals` can be used to claim all pending withdrawals initialized by the user. This function is called on each user interaction to ensure that all pending withdrawals are claimed. A special function `shutdown` can be used to register a single withdrawal request initiated by the delegator operator when undelegating the isolated Vault in EigenLayer. In addition to the general functions, the `EigenLayerWithdrawalQueue` implements functions:

- `getAccountData`: Returns claimable assets and indices of withdrawals and transferred withdrawals of a user.
- `getWithdrawalRequest`: Returns the withdrawal request of a user.

### 2.2.5 Periphery

**Claimer.** The `Claimer` is a peripheral contract that batches claiming claimable assets from withdrawal queues on behalf of a user.

**ETH Wrapper.** The ETH wrapper implements `EthWrapper.deposit()` which allows inputting ETH, WETH, stETH or wstETH to convert it to wstETH to deposit it into an ERC4626 compatible vault (typically the multivault). Note that the ETH Wrapper has been part of previous scopes.

**Whitelisted ETH Wrapper.** Inherits from the ETH wrapper but extends the functionality to allow only whitelisted users to deposit. Note that a vault's `SET_DEPOSIT_WHITELIST_ROLE` and `SET_DEPOSITOR_WHITELIST_STATUS_ROLE` can also `WhitelistedEthWrapper.setDepositWhitelist()` and `WhitelistedEthWrapper.setDepositorWhitelistStatus()`, respectively on the whitelisted ETH wrapper. WETH and ETH are the only supported assets.

**Migrator.** The migrator migrates from the simple LRT implementation (`MellowSymbioticVault` or `MellowVaultCompat`) to the multi vault implementation. The migration is implemented as follows:

1. The owner of the proxy admin contract calls `Migrator.stageMigration` to stage the migration.
2. The owner of the proxy admin contract transfers the ownership to the migrator contract with `ProxyAdmin.transferOwnership`. Note that ownership will be returned after completion or cancellation.
3. The owner can cancel the migration if necessary with `Migrator.cancelMigration`.
4. Once the migration delay has passed, the owner calls `Migrator.executeMigration` which upgrades the proxy implementation to the MultiVault and initializes the state with the expected values from the previous version and adds the expected subvault.

**EigenLayerFactoryHelper.** Helper contract for pre-computing addresses of contracts deployed by `IsolatedEigenLayerVaultFactory`.

### *2.2.6 Trust Model and Roles*

The following roles are only defined for the multivaults:

- `DEFAULT_ADMIN_ROLE`: **Fully trusted.** Has all the capabilities of the roles below.
- `SET_LIMIT_ROLE`: Trusted to set reasonable limits. Note that the limit can be interpreted as a risk parameter - especially given the buffering mechanism.
- `PAUSE_WITHDRAWALS_ROLE`: Trusted to behave reasonably and to not pause withdrawals without reason.
- `UNPAUSE_WITHDRAWALS_ROLE`: Trusted to behave reasonably and to not unpause withdrawals in case of emergency.
- `PAUSE_DEPOSITS_ROLE`: Trusted to behave reasonably and to not pause deposits without reason.
- `UNPAUSE_DEPOSITS_ROLE`: Trusted to behave reasonably and to not unpause deposits in case of emergency.
- `SET_DEPOSIT_WHITELIST_ROLE`: Trusted to behave reasonably and to not break any integrations or similar.
- `SET_DEPOSITOR_WHITELIST_STATUS_ROLE`: Trusted to behave reasonably and to not break any integrations or similar.
- `ADD_SUBVAULT_ROLE`: **Fully trusted.** Adding a malicious subvault could allow the role to drain the protocol.
- `REMOVE_SUBVAULT_ROLE`: **Fully trusted.** Removing a subvault could allow devaluing shares. As a consequence, cheap buy-ins are possible. In combination with holding the role that allows adding, the protocol could be drained.
- `SET_STRATEGY_ROLE`: Highly trusted. A malicious strategy could lead to DoS of operations. As a consequence, this role is highly trusted.
- `SET_REWARDS_DATA_ROLE`: Trusted to set up the parameters correctly. Can take all rewards.
- `REBALANCE_ROLE`: Trusted to call the rebalance as needed.
- `SET_DEFAULT_COLLATERAL_ROLE`: **Fully trusted.** Setting a bad default collateral can lead to DoS. Setting a bad collateral can lead to a loss of funds.

- `SET_ADAPTER_ROLE`: **Fully trusted.** Setting a malicious adapter can manipulate all operations, leading to a loss of funds.
- Proxy owner: **Fully trusted.** Can drain the protocol through malicious upgrades.

Note that addresses with pausing capabilities lose their rights in case they pause the respective functionality.

The following roles are only defined for the ratio strategy:

- `RATIOS_STRATEGY_SET_RATIOS_ROLE`: Trusted to set reasonable values.

Otherwise, the following roles and expectations exist:

- Users: Untrusted.
- Whitelisted addresses are typically trusted.
- Tokens are trusted to be typical ERC-20 tokens (excluding tokens with fees, and the common reentrant token types).
- The setup is trusted. Meaning, references to protocol contracts are the expected values (e.g. claimer). Similarly, all other parts of the setup are expected to be performed correctly (e.g. subvaults).

### 2.2.7 Changelog

**Version 7:**

- `Migrator` and `EigenLayerFactoryHelper` added.

**Version 5:**

- Decreased the maximum number of withdrawal requests from 50 to 14 and allowed claiming all claimable withdrawal requests at once, instead of processing only 5 at a time.

**Version 4:**

- Deployed the contracts `EigenLayerWithdrawalQueue`, `EigenLayerWstETHWithdrawalQueue`, `SymbioticWithdrawalQueue`, `IsolatedEigenLayerVault`, `IsolatedEigenLayerWstETHVault`, `IsolatedEigenLayerVaultFactory`, and `SymbioticAdapter` behind TransparentUpgradeableProxy contracts.
- Removed the contract `IsolatedEigenLayerWstETHVaultFactory` and now deploy the `IsolatedEigenLayerWstETHVault` from a second `IsolatedEigenLayerVaultFactory` using another singleton contract.
- Added logic to handle events where the operator undelegates the Vault. The function `stakeAt` now reverts in such cases until the function `shutdown` in `EigenLayerWithdrawalQueue` is called to process the withrawal of undelegated stake.
- Modified the function `transferPendingAssets` to be able to transfer claimable assets. However, the `MetaVault` claims assets before sending pending assets, so with the current `RatiosStrategy` in place, it never sends assets that were previously claimable.

**Version 3:**

- `MultiVault.maxDeposit()` and `MultiVault.assetsOf()` have been removed. `MultiVault.assetsOf()` returned the claimable, the pending and subvault balances for a given subvault/queue pair. As of version 3, the amounts must be queried individually and cannot be queried in a batch. `MultiVault.maxDeposit()` used to be the central entry-point for querying maximum deposits. However, now adapters are queried directly.
- Prior to version 3, adapters were not allowed to give out approvals to the subvault as the multivault performs that. However, now they must.

- Note that before version 3 there was no notion of withdrawal pauses. Consequently, the withdrawals did not revert due to the ratio strategy in case withdrawals were unsupported.
- The `WhitelistedEthWrapper` has been added to the scope in this version and has been added accordingly to the system overview.

**Version 2:**

- Renamed `MultiVault.maxWithdraw()` and `Adapter.maxWithdraw()` functions to `MultiVault.assetsOf()` and `Adapter.stakedAt()`, respectively.
- Renamed `Adapter.validateFarmData()` to `Adapter.validateRewardData()`.
- Renamed `SHARES_STRATEGY_SET_RATIO_ROLE` to `RATIOS_STRATEGY_SET_RATIOS_ROLE`.

# 3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

# 4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

| **Likelihood** | **Impact** | | |
|---|---|---|---|
| | High | Medium | Low |
| High | **Critical** | **High** | **Medium** |
| Medium | **High** | **Medium** | **Low** |
| Low | **Medium** | **Low** | **Low** |

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

# 5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 1 |

- Bank Run on Excess Funds in Vault Prior to Slashing Event Risk Accepted

| | |
|---|---|
| Low-Severity Findings | 6 |

- EigenLayer Withdrawals Attempted When Maximum Is Reached Acknowledged
- Manipulateable Withdrawal Pauses for ERC-4626 Risk Accepted
- Overly Sensitive Reverting in Ratios Strategy Acknowledged
- Improper ERC-4626 Valuation Risk Accepted
- ERC-4626 Violations Acknowledged
- Problems With Configurations Code Partially Corrected Acknowledged

## 5.1 Bank Run on Excess Funds in Vault Prior to Slashing Event

Design Medium Version 1 Risk Accepted

*CS-MLW-MULTI-002*

The vault holds liquid assets when not all funds can be deposited into external protocols. The idle assets can be used to convert shares for underlying assets at the current exchange rate and withdraw them from the vault without any delay.

This can lead to a bank run on the excess funds in the vault before a slashing event occurs. Note that slashing events are predictable (e.g. `VetoSlasher` in Symbiotic). Consequently, the remaining share owners are the only ones affected by the slash.

Similarly, with multiple subvaults in use, pending assets of one of the slashed subvault can be worth less than the assets of another subvault. This can lead to a bank run on the pending assets in the vault before a slashing event occurs, and it incentivizes frontrunning / backrunning withdraw transactions of the other user since they are not able to decide what kind of (pending) assets to receive from the vault.

---

**Risk accepted:**

Mellow Finance is aware of the issue and accepts the risk.

## 5.2 EigenLayer Withdrawals Attempted When Maximum Is Reached

Correctness Low Version 5 Acknowledged

*CS-MLW-MULTI-038*

As of version 5, the number of EigenLayer withdrawals is limited to 14 (with the exception of shutdown where it may be 15). However, note that the `RatioStrategy` is unaware of the limitation (e.g. withdrawals are not treated as paused). Thus, the Vault withdrawals could be temporarily DoSed due the impossibility of creating additional requests.

## 5.3 Manipulateable Withdrawal Pauses for ERC-4626

Design Low Version 3 Risk Accepted

*CS-MLW-MULTI-015*

Note that only ERC-4626 with withdrawal pauses are allowed (no other withdrawal limitations). As a consequence, the withdrawal pause is determined with `ERC4626.maxRedeem()`.

However, to prevent unnecessary reverts, the protocol is only treated as paused if the share balance of the vault is non-zero. However, that check is easily manipulateable through donations and malicious users could render the check ineffective under certain circumstances. Ultimately, an unwanted temporary DoS scenario could occur for users.

---

**Risk accepted:**

Mellow Finance is aware of the issue and accepts the risk.

## 5.4 Overly Sensitive Reverting in Ratios Strategy

Design Low Version 3 Acknowledged

*CS-MLW-MULTI-016*

`RatiosStrategy.calculateState()` will revert when withdrawing or rebalancing if an adapters returns `true` for `areWithdrawalsPaused()`. However, note that this might be overly restrictive.

For example:

1. A withdrawal action might not necessarily withdraw from a subvault. It could be that the vault is underallocated and that thus withdrawals will be pulled from other subvaults which could have sufficient funds.
2. Similarly, during a rebalance a subvault where withdrawals are paused might receive a deposit. However, the strategy would revert.

Note that there are other scenarios where pausing would not affect execution due to not withdrawal happening for a given subvault.

Ultimately, the strategy could be overly sensitive to withdrawal pauses.

---

**Acknowledged:**

Mellow Finance is aware of the issue and acknowledges it.

## 5.5 Improper ERC-4626 Valuation

Design Low Version 2 Risk Accepted

*CS-MLW-MULTI-017*

Note that the `previewRedeem` function will take current on-chain conditions into account, and hence the exchange rate will vary. For example, the vault might simulate a swap. If the liquidity is low, the vault will quote a lower price for the shares, allowing a new user to buy in at a lower price.

The ERC-4626 specification:

```
Note that any unfavorable discrepancy between convertToAssets and previewRedeem SHOULD be considered slippage in share price
or some other type of condition, meaning the depositor will lose assets by redeeming.
```

---

**Risk accepted:**

Mellow Finance specified that only tokens where `token.previewRedeem(token.balanceOf(multiVault))` cannot be manipulated will be used. While that resolves the issue, the `ERC4626.convertTo*` functions are more suitable for such usage.

## 5.6 ERC-4626 Violations

Design Low Version 1 Acknowledged

*CS-MLW-MULTI-005*

The contracts implement the ERC-4626 tokenized vault standard. However, the standard is violated.

Namely, that is due to the withdrawals not being instant. Hence, the withdrawn amount will only be received only partially. The remainder will be received as shares. Additionally, the respective event will be consequently inaccurate.

Function `totalAssets` and all other functions relying on total assets (e.g., `convertToShares`, `convertToAssets`, `maxDeposit`, `maxWithdraw`, `maxMint`, `maxRedeem`) must not revert according to ERC-4626. However, in the case of ERC-4626 adapters, the function `totalAssets` calls `ERC4626Adapter.stakedAt`, which in turn calls `ERC4626.previewRedeem`. previewRedeem may revert when a call to redeem would revert.

---

Version 3:

The function `RatiosStrategy.calculateState` reverts when funds are withdrawn from the protocol and one of the ERC4626 sub-vaults is paused, as detected by `areWithdrawalsPaused`. However, the functions `maxWithdraw` and `maxRedeem` do not account for these withdrawal restrictions and do not return 0 as required by ERC4626.

---

**Acknowledged:**

Mellow Finance is aware of the issue and acknowledges it.

## 5.7 Problems With Configurations

Design | Low | Version 1 | Code Partially Corrected | Acknowledged

*CS-MLW-MULTI-006*

The configuration of the protocol is of utmost importance. Incorrect configuration might lead to a loss of funds and similar.

Below is a list of potential problems when suitable contracts are used (excludes setting unsuitable addresses):

1. `MultiVault.removeSubvault()`: Removing a subvault could create a loss of funds. If the subvault is to be readded, this creates an arbitrage opportunity where attackers could use the artificially lowered share price to buy shares cheaper.
2. `MultiVault.set*Adapter()`: In case an adapter is updated, that may lead inconsistencies of withdrawal queue storage (multivault tracks another withdrawal queue than the adapter). In case a subvault is removed and readded, the `handleVault()` function could return a new queue. That could lead to losses in pending and claimable assets. Similarly, that could have an effect on `Claimer`.
3. `MultiVault.setDefaultCollateral()`: Given that the default collateral address cannot be changed, the suitability in regards to the asset should be checked. Similarly, its suitability should be checked against the expected default collateral factory.

Further, there might be other potential helpful sanity checks preventing configuration errors:

4. Validating addresses against factories and registries. For example, each symbiotic vault and reward could be validated against factories. Similarly, that is true for the default collaterals.
5. Validating the ratios of `RatioStrategy`: While it makes sense to define that the sum of maximum ratios can exceed 100%, allowing that for minimum ratios is expected to be not meaningful.

However, note that implementing these checks does not prevent setups that are not meaningful or compatible.

---

1. *Acknowledged:* Mellow Finance is aware of the issue and acknowledged it. Mellow Finance mentioned that for subvaults with significant value, this will only be called under a protocol-level lock.
2. *Acknowledged:* Mellow Finance is aware of the issue and acknowledged it. Additionally, Mellow Finance mentioned that the functions will not be called after the intial adapters are set, execept a migration level update is necessary (e.g. EigenLayer introducing slashing).
3. *Corrected:* The underlying token is validated. However, the default collateral is not validated against the factory.
4. *Partially corrected:* The Symbiotic adapter now validates the vaults against factories. However, other factory checks are not performed.
5. *Acknowledged:* Mellow Finance acknowledged the issue but prefers the flexibility.

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |

| | |
|---|---|
| High-Severity Findings | 2 |

- Pending Assets Become Claimed During Withdrawal Code Corrected Specification Changed
- Incorrect Valuation of ERC-4626 Code Corrected

| | |
|---|---|
| Medium-Severity Findings | 6 |

- EigenLayer Queue wstETH Mix-Up Code Corrected
- Isolated EigenLayer Vault Not Functional Code Corrected
- Operator Undelegations Are Not Accounted for Specification Changed Code Corrected
- Transfer Pending Assets Skips Assets Code Corrected
- Escalation of Privileges Code Corrected
- Insufficient Limitations for Strategies Specification Changed

| | |
|---|---|
| Low-Severity Findings | 13 |

- Claimable Assets Are Transferred as Pending Code Corrected
- Shutdown Does Not Stop Deposits Code Corrected
- Temporary Vault DoS Due to Shutdown Delay Code Corrected
- Double Tracking of Withdrawals and Transferred Withdrawals Code Corrected
- EigenLayer Not Ready to Be Integrated With Without Upgrades Code Corrected
- Implicit Minimum Deposit for EigenLayer Code Corrected
- Incorrect Manager for Pausing Check for EigenLayer Code Corrected
- getOrCreate Never Gets Code Corrected
- Incorrect Maximum Computation in Ratios Strategy Code Corrected
- Incorrect Maximum Deposit Computation in Ratio Strategy Code Corrected
- Incorrect Target for ERC-4626 Maximum Deposit Query Code Corrected
- Old Vault Contracts Can Not Be Migrated Easily Code Corrected
- Sending Pending Assets to Yourself Leads to Losses Code Corrected

| | |
|---|---|
| Informational Findings | 11 |

- Insufficient Validation in Migrator Code Corrected
- Gas Optimizations Code Corrected
- Incomplete Deletion of Shares Code Corrected
- Inconsistent Event Argument Amount in Event Transfer Code Corrected
- Incorrect Argument Order for Key Computation Code Corrected

- Return Value Not Used Specification Changed
- Unreachable Adapter Code Code Corrected
- Giving Potentially Unnecessary Approvals Code Corrected
- Implementation Contract Can Be Initialized Code Corrected
- Lack of Events Code Corrected
- Misleading Names of Functions and Variables Code Corrected

## 6.1 Pending Assets Become Claimed During Withdrawal

Design High Version 3 Code Corrected Specification Changed

*CS-MLW-MULTI-020*

The EigenLayer withdrawal queue can claim 5 withdrawals in each call. If there are more than 5 withdrawals claimable in EigenLayer, only the first 5 are accounted as claimable, and the others are accounted as pending until they move into one of the first 5 slots.

The function `handleWithdrawals` (called internally by `request`, `transferPendingAssets`, and `claim`) processes up to 5 withdrawals at a time. As a result, assets that were previously pending in the queue can become claimed.

However, the MultiVault can not account for the fact that pending assets can become claimed during the withdrawal process and hence operates on stale data provided by the `RatiosStrategy`. The vault follows these steps:

1. Estimate values for claimable and pending assets from the queue.
2. Call into the queue to claim assets by calling `claim`, which calls `handleWithdrawals`.
3. Call into the queue to transfer pending assets by calling `transferPendingAssets`, which also calls `handleWithdrawals`.

The two calls to `handleWithdrawals` can make assets claimable and hence cause a discrepancy between the estimated and actual values of claimable and pending assets. For example, let's say there are 7 withdrawals (A, B, C, D, E, F, G) that are all claimable in EigenLayer. Then the `RatiosStrategy` will estimate the following values:

- claimableAssetsOf = A + B + C + D + E
- pendingAssetsOf = F + G

At the start of the withdrawal, the RatiosStrategy calculates F + G for pending assets. However, in the function call to `claim`, the first 5 withdrawals are claimed. In the subsequent call to `transferPendingAssets`, the function `handleWithdrawals` will now claim the remaining 2 withdrawals.

Now, there are not enough pending assets available to send out and function `transferPendingAssets` reverts with the message "EigenLayerWithdrawalQueue: insufficient pending assets".

This can cause a denial of service (DOS) in key functions (rebalancing, withdrawals, etc.) in the MultiVault system and force the system to upgrade.

---

Version 4:

The code has been partially corrected. Namely, a function `_transferClaimableAsPending` has been introduced to transfer funds made claimable due to the double `handleWithdrawals` computation.

**However, more issues persist:**

1. `_transferClaimableAsPending` performs the following computation:

   ```
   if (assets < accountDataFrom.claimableAssets) {
       revert("EigenLayerWithdrawalQueue: insufficient pending assets");
   } else {
       accountDataFrom.claimableAssets -= assets;
       accountDataTo.claimableAssets += assets;
   }
   ```

   However, if `assets` is <= than `claimableAssets`, the transfer can succeed.

2. A refactoring change lead to the iteration of `handleWithdrawals` being changed. Namely, the array to iterate over was loaded into memory and then iterated over. Now, the iteration happens on storage. However, given the pop and swap nature of the enumerable set, the order will change. While it is accounted that not items are skipped, a problem is created due to the reordering of the underlying array. More precisely, assuming the 7 withdrawal from above (A, B, C, D, E, F, G), the initial version claimed the following items (A, B, C, D, E) while the new implementation claimed the items (A, G, F, E, D). As a consequence, the iteration of `claimableAssets` and `pendingAssets` severely mismatches the iteration of `handleWithdrawals`.
3. Note that the combination of `handleWithdrawals` leads to a mismatch in iteration, too - even in the previous version (the underlying array for the second handling of withdrawals would have been (G, F)). Note that even if issue 2 were not present, the swap and pop inside the first iteration will change the order in the second iteration. Ultimately, the order will not be the same as `claimableAssets` and `pendingAssets`, leading to potential reverts.

Note that it could be worth considering that a set might not be an optimal data structure for the self-requested withdrawals: alternatives for less code complexity could be list-like structures such as dequeue. Further, these could help preserve the ordering according to `startBlock` so that iterations can exit early. *However*, these might not be fully suitable due to the insertion of items when acceptancing transferred request shares and `contains` being inefficient. However, that problem can be worked around.

**Code corrected:**

In **Version 5**, the remaining issues have been fixed:

1. The sign in the comparison have been flipped to ensure that the assets are larger than the claimable assets.
2. The array is now loaded into memory again, so changes to the order of the array in storage do not affect the iterations.
3. The code has been greatly simplified: The function `handleWithdrawals` no longer processes up to 5 withdrawals at a time. Instead, it processes all claimable withdrawals in the queue (up to 14). The separation between claimable and pending assets is now purely based on the timestamp of the withdrawal, which does not change between multiple calls to `handleWithdrawals`.

## 6.2 Incorrect Valuation of ERC-4626

**Security** **High** **Version 1** **Code Corrected**

*CS-MLW-MULTI-001*

The adapter's `maxWithdraw()` function returns the value held by the vault for a given ERC-4626 token. The `ERC4626Adapter` incorrectly valuates the ERC-4626 balance.

More specifically, the adapter computes `maxWithdraw()` by calling `ERC4626.maxWithdraw()`. However, that does not correspond to the value but to the value that can currently be withdrawn. As a consequence, the valuation of a MultiVault may be incorrect.

For example, consider a scenario where the ERC-4626 is paused. Consequently, the valuation will then return 0 which could undervalue the position significantly.

Ultimately, the incorrect implementation may lead to low-value buy-ins.

Further, note that not only is the `maxWithdraw()` the improper way of estimating the value held but the argument passed to it is incorrect (the token is passed instead of the vault). As a consequence, the oracle will be always incorrect.

---

**Code corrected**:

The meta vault has been updated to query the function `ERC4626Adapter.stakedAt()` instead of `ERC4626Adapter.maxWithdraw()` to retrieve the value held by the Vault. The adapters `stakedAt()` function simulates redeeming all shares owned by vault with `ERC4626.previewRedeem()`.

```
function stakedAt(address token) external view returns (uint256) {
    IERC4626 token_ = IERC4626(token);
    return token_.previewRedeem(token_.balanceOf(vault));
}
```

Note that this fix resulted in issue Improper ERC-4626 valuation.

## 6.3 EigenLayer Queue wstETH Mix-Up

Correctness Medium Version 3 Code Corrected

*CS-MLW-MULTI-021*

All currency computations of queue's should be denominated in the underlying token of the vault. However, due to stETH being used on EigenLayer instead of wstETH as in the multi vault and isolated vault, a mix-up of the two currencies is created in the `EigenLayerWithdrawalQueue`.

When creating a withdrawal request through `request` withdrawals are tracked. Note that the function is only called by the isolated vault. In case of wstETH being the multi-vault's underlying, a conversion from wstETH to stETH is performed so that the `request.assets` is denominated in stETH.

In contrast, `transferPendingAssets`, where queue shares are transferred according to the input amount of underlying, does not account for wstETH/stETH conversion. That is due to the multi vault specifying the parameter `amount` in wstETH. However, the local variable `accountAssets` is denominated in stETH. The later comparison of the two values compares two distinct currency units. Ultimately, that might lead to incorrect accounting.

Similarly, other functions such as `pendingAssetsOf` and `claimableAssetsOf` could return incorrect results.

---

**Code corrected:**

In Version 4 the issue had been partially resolved and a confusion of wstETH and stETH was possible while an issue was introduced due to the possibility of off-by-one errors of stETH transfers (see Issue #442 on Lido's Github). The code has been corrected in Version 5. Namely, the isolated vault is now only uses wstETH and shares for accounting. Further, it does not interract with strategies directly but rather queries `sharesToUnderlyingView` and `underlyingToSharesView` through the vaults that are specialized.

## 6.4 Isolated EigenLayer Vault Not Functional

**Correctness** **Medium** **Version 3** **Code Corrected**

*CS-MLW-MULTI-022*

The `EigenLayerAdapter` approves the `IsolatedEigenLayerVault` to allow pulling funds. However, the isolated vault does not pull funds from the multi-vault. Additionally, the isolated vault does not approve EigenLayer's `StrategyManager` and thus no funds will be pullable by EigenLayer.

As a consequence, `IsolatedEigenLayerVault.deposit` is unusable, and no deposits can be made to EigenLayer.

---

**Code corrected:**

The code has been adjusted to pull funds and to approve the `StrategyManager`.

## 6.5 Operator Undelegations Are Not Accounted for

**Design** **Medium** **Version 3** **Specification Changed** **Code Corrected**

*CS-MLW-MULTI-023*

The Vault delegates its stake to an operator on EigenLayer. Note that operator (or their delegation approver) is allowed to undelegate any staker by calling Eigenlayer's `DelegationManager.undelegate` function. This queues a withdrawal of the operated stake. However, since this withdrawal is not stored in the withdrawal queue, the multi vault is not aware of these withdrawals and ignores them. This can lead to a discrepancy between the actual and the reported assets of the Vault until the Vault is upgraded.

---

**Code corrected changed:**

In Version 4, the function `EigenLayerAdapter.stakedAt` reverts if the isolated vault is not delegated to any operator and the withdrawal queue is not shut down.

```
if (
    !delegationManager.isDelegated(isolatedVault)
        && !IEigenLayerWithdrawalQueue(withdrawalQueue).isShutdown()
) {
    revert("EigenLayerAdapter: isolated vault is neither delegated nor shut down");
}
```

This prevents any user from interacting with the vault as soon as the operator undelegates the vault, thereby stopping any vault operation that could exploit the mispricing.

To resolve the issue, anyone can call the function `EigenLayerWithdrawalQueue.shutdown` with the withdrawal data of the undelegation event that is validated with Eigenlayer and create a new request. This sets the *isShutdown* flag to true and the `stakedAt` function no longer reverts.

The Mellow Finance describes the following steps to offboard the Vault:

> For future mitigation of this event, it is expected that once all funds have been fully withdrawn from this isolated vault, we will remove this subvault from the corresponding MultiVault.

## 6.6 Transfer Pending Assets Skips Assets

Design Medium Version 3 Code Corrected

*CS-MLW-MULTI-024*

The function `EigenLayerWithdrawalQueue.transferPendingAssets` caches the number of all pending assets of an account and then loops over all the withdrawals by index to transfer them to the beneficiary, deleting the withdrawal indices from the list of withdrawals of the user if the full amount is transferred. The code uses library function `EnumerableSet.remove` that deletes elements via "swap and pop" - changing the order of the list.

Example:

1. Initial state:
   - The loop starts with the first withdrawal, which has an index of 103.
   - The list of withdrawals is [103, 106, 111].
2. The condition *accountAssets <= amount* is true for the withdrawal at index 103. Therefore, the function removes this entry using the pop and swap method.

   This method removes the element at index 103 and replaces it with the last element in the list, which is 111. The list is then shortened by one element.
3. Updated state:
   - The loop moves to the next withdrawal index, which is now 106.
   - The updated list of withdrawals is [111, 106].

As seen in the example, the withdrawal index *111* is skipped.

Additionally, the function can try to access elements of *withdrawals* that are out of bounds of the array since we do not take into account that the array is shrunk in the loop.

In summary the bug causes the transfer of pending assets to revert, preventing withdrawals and rebalancing.

---

**Code corrected:**

In Version 4, whenever an entry in *withdrawals* is removed, the index of the loop is not incremented, and the total number of withdrawals `pendingWithdrawals` is decremented. This ensures that removing the element with the pop and swap method does not skip any withdrawals or cause the loop to attempt to access elements that are out of bounds.

## 6.7 Escalation of Privileges

Security Medium Version 1 Code Corrected

*CS-MLW-MULTI-003*

While many roles are fully trusted due to the design of the protocol, the trust in the `SET_REWARDS_DATA_ROLE` is limited. Addresses holding that role should not be able to affect user funds. However, addresses holding that role can drain the protocol.

Consider the following scenario:

1. The malicious address holding the role adds a malicious farm where the curator fee is 0% and the distribution farm is the malicious address. Note that the farm contract is a malicious contract implementing the required interface. The reward token is any of the underlying tokens held by the protocol.
2. The malicious address pushes the rewards for the farm ID.
3. As a consequence, the rewards are claimed through the trusted adapter set up by a fully trusted role. Note that adapters typically validate farm data only to a limited degree and would typically allow claiming rewards from arbitrary contracts.
4. Now, the malicious farm contract takes a flashloan and creates a huge deposit.
5. As a consequence, the malicious farm contract receives multivault shares.
6. Eventually, the claim finalizes without any needed transfer.
7. However, the multivault will see a balance delta and move the funds to the curator address which is the malicious address.
8. Ultimately, the vault shares can be redeemed.

In combination with the `SET_STRATEGY_ROLE` or any other privileged role on strategies, the attack could be further escalated.

Note that the privilege escalation can lead to the multivault being completely drained.

---

**Code corrected:**

A reentrancy lock preventing the attack is specified.

Additionally, the role is fully trusted now and limiters on supported reward tokens have been put in place.

## 6.8 Insufficient Limitations for Strategies

Design Medium Version 1 Specification Changed

*CS-MLW-MULTI-004*

The protocol assumes that external protocols can only limit the maximum deposits. However, this assumption in practice is not sufficient. For example, the ERC-4626 adapter directly is violating the assumption.

While generally arbitrary values could be disabled, it might make sense to consider deposit and withdrawal ranges, given that most protocols typically limit in ranges.

Consider the list below:

- Withdrawal upper bound: ERC-4626 has a maximum withdrawal amount. The protocol will not respect such bounds. Due to that reverts on rebalancing and withdrawals could occur and DoS said functionalities.
- Deposit lower bound: Some protocols enforce minimum deposits (e.g. Stader). These are not uncommon to prevent dusty deposits. In case there was a minimum deposit amount, the protocol could be DoSed in depositing and rebalancing.
- Withdrawal lower bound: Some protocols enforce minimum withdrawal amounts (e.g. Stader). These are not uncommon. In case there was a minimum withdrawal amount, rebalancing and withdrawals could be DoSed.

Further, note that pausing functionality is common across a wide range of protocols. Pausing certain functionality might lead DoS of the multivault architecture.

To summarize, the lack of maximum withdrawal amounts violates existing integrations. The lack of lower bounds might prevent from integrating with a wide range of additional protocols. Note that additionally, that makes the related ERC-4626 getter functions incorrect, too.

---

**Specification changed:**

Mellow Finance restricts certain kind of protocols. Namely, protocols with the bounds above are not supported.

However, new functionality was added to support protocols with pauses.

## 6.9 Claimable Assets Are Transferred as Pending

Correctness Low Version 5 Code Corrected

*CS-MLW-MULTI-039*

In Version 4, subsequent calls to *handleWithdrawals* could make assets claimable and cause a discrepancy between the estimated and actual values of claimable and pending assets. This required the introduction of a function *_transferClaimableAsPending* to transfer assets that were previously pending and now made claimable due to a call to *handleWithdrawals*.

However, that issue was resolved in Version 4 by considering all pending assets that are claimable as claimable assets instead of just the first 5. See *Pending Assets Become Claimed During Withdrawal*.

Despite this, function `EigenLayerWithdrawalQueue.transferPendingAssets` still calls the function `_transferClaimableAsPending` if more pending assets are requested for transfer than are available. This allows a user to send the sum of pending and claimable assets with the function `transferPendingAssets`, which can be surprising to the user and is not documented.

The Vault requests withdrawals with values retrieved from the `RatioStrategy`. While the current implementation of the `RatioStrategy` is correct and does not request to send claimable assets as pending, a new buggy Strategy contract could have a flaw in the calculation that could get abused by users to receive claimable instead of pending assets.

---

**Code corrected:**

The respective code has been removed.

## 6.10 Shutdown Does Not Stop Deposits

Design Low Version 4 Code Corrected

*CS-MLW-MULTI-036*

If an undelegation for the isolated vault is initiated by an external party, the protocol blocks operation with the Vault by reverting the function `EigenLayerAdapter.stakedAt`. Note that this reverts:

- User operations
- Rebalancing operations (due to `RatioStrategy` depending on the reverting function)

To resolve the reverts, any address can call the function `EigenLayerWithdrawalQueue.shutdown` to register the withdrawal from the aforementioned undelegation.

Once the queue is shutdown, the subvault should be offborded so that no further deposits can be made. However, this does not hold true since `EigenLayerAdapter.maxDeposit` does not return zero during when the queue is shut down. While these funds can be withdrawn through regular withdrawals, the stake in strategies without an operator provides no economic security to any service and do not generate any yield.

To summarize:

1. Once a queue is shut down, deposits to the corresponding subvault can be made.
2. Additionally, in case a hypothetical strategy not depending on `stakedAt` were to be used, it could be possible that successful rebalances could occur. Such a case could be worth exploring.

---

**Code corrected:**

In Version 5 of the code, `EigenLayerAdapter.maxDeposit` returns zero when the queue is shut down. This ensures that no further deposits can be made to the subvault.

## 6.11 Temporary Vault DoS Due to Shutdown Delay

Design Low Version 4 Code Corrected

*CS-MLW-MULTI-037*

The `EigenLayerWithdrawalQueue.shutdown` function pushes requests to the set of self-requested withdrawals. Note that the following code implements this:

```
if (isSelfRequested) {
    if (accountData.withdrawals.length() + 1 > MAX_PENDING_WITHDRAWALS) {
        revert("EigenLayerWithdrawalQueue: max withdrawal requests reached");
    }
    accountData.withdrawals.add(withdrawalIndex);
```

In case the withdrawals are of size `MAX_PENDING_WITHDRAWALS`, the shutdown function will revert, leading to a temporary DoS for the full vault due to pricing of shares reverting. In case all the withdrawals were recent, the waiting times will be maximized.

Ultimately, the shutdown could be blocked for a period of time so that vault interactions remain DoSed.

---

**Code corrected:**

The code has been adjusted to allow exceeding the maximum size for the shutdown.

## 6.12 Double Tracking of Withdrawals and Transferred Withdrawals

Correctness Low Version 3 Code Corrected

*CS-MLW-MULTI-025*

The `EigenLayerWithdrawalQueue` tracks pending withdrawals either as self-requested withdrawals or transferred withdrawals to prevent spamming users with small withdrawals.

Withdrawals enter the transferred withdrawals if the request created is not self-requested or in case of a queue shares transfer with `transferPendingAssets`. The latter allows for double-tracking a request in both the self-requested and the transferred withdrawals. Consider the following scenario:

1. Alice has a withdrawal with x shares and transfers `x-1` shares to Bob.
2. Bob accepts the withdrawal and transfer `x-1` shares to Alice.

At this point, Alice has x shares but tracks the withdrawal in both sets.

Note that the underlying issue is that a withdrawal on transfer should not always be added to the set of transferred withdrawals but only if the withdrawal is not contained in the set of self-requested withdrawals.

Generally, the double-tracking is handled in `_handleWithdrawal` and `acceptPendingAssets`. However, DoS problems could arise nevertheless. Namely, if Alice transferred `x` shares to Charlie, `transferPendingAssets` would only remove the item from the self-requested withdrawals. If Alice, accepts the transferred withdrawal she could end-up having a zero-shares withdrawal tracked inside of her withdrawals set which could lead to unremovable withdrawals due to the `accountShares == 0` check in `_handleWithdrawal`.

---

**Code corrected:**

`transferPendingAssets` only adds to the not self-requested withdrawals if there is no entry in the self-requested withdrawals. Ultimately, not double tracking can occur.

## 6.13 EigenLayer Not Ready to Be Integrated With Without Upgrades

Design | Low | Version 3 | Code Corrected

*CS-MLW-MULTI-026*

While contracts are already deployed, EigenLayer is still in the process of development. Namely, that is due to the slashing logic not being finalized yet. Due to that, deploying the EigenLayer contracts in a non-upgradeable fashion is strongly discouraged. For example, PR 679 for the EigenLayer contracts severily changes the integration, breaking assumptions made during the audit.

Note that while many parts of the system are upgradeable the queue is not and thus this could lead to loss of funds.

---

**Code corrected:**

All the contracts are now upgradeable. That includes `EigenLayerWithdrawalQueue`, `EigenLayerWstETHWithdrawalQueue`, `SymbioticWithdrawalQueue`, `IsolatedEigenLayerVault` and `IsolatedEigenLayerWstETHVault`.

## 6.14 Implicit Minimum Deposit for EigenLayer

Correctness | Low | Version 3 | Code Corrected

*CS-MLW-MULTI-027*

While EigenLayer has no explicit minimum deposits, it enforces that at least one share must be minted which creates an implicit minimum requirement. Note that this violates note Unsupported Limits in Integrations.

Note that the isolated EigenLayer vaults do not handle this gracefully. While the wstETH isolated aims to resolve the issue, the regular isolated vault does not. However, the check in the wstETH vault is also insufficient:

```
if (assets <= 1) {
    // insignificant amount
    return;
}
```

Namely, that is due to the `underlyingToSharesView` function returning 0 in other cases, too. It's computation can be summarized to

```
(amountUnderlying * (totalShares + 1000)) / (_tokenBalance() + 1000);
```

When for example 2 assets are deposited, then this will compute the following:

```
(2*totalShares + 2000) / (_tokenBalance() + 1000);
```

That will return zero if

```
2*totalShares + 1000 < _tokenBalance()
```

Note that this could be the case when the strategy has created significant yield.

---

**Code corrected:**

The code has been adjusted to perform the following computation:

```
if (IStrategy(strategy).underlyingToSharesView(assets) == 0) {
    // insignificant amount
    return;
}
```

## 6.15 Incorrect Manager for Pausing Check for EigenLayer

Correctness Low Version 3 Code Corrected

*CS-MLW-MULTI-028*

`EigenLayerAdapter.areWithdrawalsPaused` defines whether an external protocol is paused to ensure that withdrawals are not attempted in such scenarios. However, instead of checking the state of the EigenLayer's `DelegationManager` the `StrategyManager` state is check for pausing leading to incorrect results.

---

**Code corrected:**

Version 4 queries the *DelegationManager* for the pausing state.

## 6.16 `getOrCreate` Never Gets

Correctness Low Version 3 Code Corrected

*CS-MLW-MULTI-029*

getOrCreate should create an isolated EigenLayer vault for a tuple (owner, operator, strategy) if it has not been created. If it had been created previously, it should return the address of the vault and the respective queue. However, the storage for isolatedVaults is never written to so that the code inside the if statement

```
if (isolatedVault != address(0)) {
    return (isolatedVault, instances[isolatedVault].withdrawalQueue);
}
```

is unreacheable. Ultimately, the function would revert.

---

**Code corrected:**

In Version 4 the function writes the address of the isolated vault to isolatedVaults.

## 6.17 Incorrect Maximum Computation in Ratios Strategy

Correctness Low Version 1 Code Corrected

*CS-MLW-MULTI-013*

The limit imposed by the MultiVault.maxDeposit() function is accounted improperly. Namely, the limit will be over-estimated.

The function RatiosStrategy.calculateState() computes minima and maxima for target allocations according to the ratios. It further considers the maximum allowed deposit as follows:

```
for (uint256 i = 0; i < n; i++) {
    (state[i].claimable, state[i].pending, state[i].staked) = multiVault.maxWithdraw(i);
    uint256 assets = state[i].staked + state[i].pending + state[i].claimable;
    totalAssets += assets;
    state[i].max = multiVault.maxDeposit(i);
    if (type(uint256).max - assets >= state[i].max) {
        state[i].max += assets;
    }
}
```

While the lack of an else branch is part of another issue, this issue focuses on the addition of multiVault.maxDeposit(i) and assets.

Namely, maxDeposit() will define how much can be added on top of the amount currently held in the subvault. Thus, the maximum target allocation should be limited to state[i].staked + multiVault.maxDeposit(i). Namely, that is due to the "staked" amount being the amount in the protocol itself. The claimable and the pending assets should not affect the vaults maximum deposit in that regard.

Consider the following scenario:

1. Assume that a hypothetical protocol that is ERC4626-like exists. However, the protocol has an unstaking mechanism where funds could be delayed. That protocol could be similar to Mellow Multivaults in that regard.
2. An adapter exists that supports that protocol. Also, a relevant subvault exists.
3. Now, the allocation is temporarily set to 50%, leading to huge withdrawals, increasing the pending assets.
4. Eventually, the allocation is increased to 100%.

5. Still, 1M tokens are still pending while 1M tokens are staked. Eventually, 1M tokens are claimable.
6. The `maxDeposit()` changes for the external protocol due to some reconfiguration of risk parameters and the protocol supports no deposits from the vault anymore.
7. The maximum computed will however be 2M (due to the claimable amount) while currently only 1M is in the protocol.
8. Ultimately, an allowed deposit delta of 1M will be identified.
9. The execution reverts.

To summarize, the maximum accounting in the strategies accounts for the maximum target allocation incorrectly, leading to partial DoS possibilities.

---

**Code corrected:**

The maximum accounting is now done by using the only `staked`.

```
state[i].max = maxDeposit + state[i].staked;
```

## 6.18 Incorrect Maximum Deposit Computation in Ratio Strategy

Correctness Low Version 1 Code Corrected

*CS-MLW-MULTI-012*

To compute the maximum target amount to reach, `RatioStrategy.calculateState()` performs the following computations.

```
state[i].max = multiVault.maxDeposit(i);
if (type(uint256).max - assets >= state[i].max) {
    state[i].max += assets;
}
```

Note that this should compute the maximum to be `multiVault.maxDeposit(i) + assets` as the upper bound imposed by the deposit limit.

However, due to overflow potentials, a specialized computation is required. Thus, the `if` branch only adds `assets` to the limit if possible. However, the `else` is missing. Namely, in that scenario the maximum should become `uint256.max`.

Consider the following example:

1. `multiVault.maxDeposit(i)` is 2.
2. `assets` is `uint256.max - 1`.
3. Now the `state[i].max` will remain 2.

While such scenarios are unlikely, they could lead to incorrect allocations.

---

**Code corrected:**

The code has been adjusted. An `else` branch has been added to set the maximum to `uint256.max`.

## 6.19 Incorrect Target for ERC-4626 Maximum Deposit Query

Correctness Low Version 1 Code Corrected

*CS-MLW-MULTI-011*

The `ERC4626Adapter` derives the amount allowed to be deposited with a call to `ERC4626.maxDeposit()`. However, not the maximum deposit for the vault is queried but the maximum deposit amount for the ERC-4626 token. Ultimately, the estimation will be incorrect.

---

**Code corrected:**

In Version 2 the adapter queries the maximum deposit of the vault.

## 6.20 Old Vault Contracts Can Not Be Migrated Easily

Design Low Version 1 Code Corrected

*CS-MLW-MULTI-007*

There are prior versions of the `Vault` that need to be migrated to `MultiVault`. There are several issues with the design of the contract that make it difficult to migrate to the new version.

1. The `MultiVault` contract requires the `initialize` function to set up storage. However, this function uses the `initializer` modifier to prevent reinitialization. Since prior versions of the vault contract have already been initialized, calling `initialize` will revert.
2. The previous vault implementation of the simple LRT defined the `SET_FARM_ROLE`. That role identifier has been replaced by `SET_REWARDS_DATA_ROLE`. As a consequence, the farm adders will lose their privileges and need to be granted the farm role. Ultimately, if the `SET_FARM_ROLE` is reintroduced to the system, problems may arise due to unwanted and forgotten addresses holding that role.

---

**Code corrected:**

1. Corrected. `reinitializer(2)` is used which is compatible with new vaults and upgrades from version 1 (simple LRT).
2. Corrected. `SET_FARM_ROLE` is now used.

## 6.21 Sending Pending Assets to Yourself Leads to Losses

Correctness Low Version 1 Code Corrected

*CS-MLW-MULTI-014*

The `SymbioticWithdrawalQueue` contract allows sending assets pending withdrawal from Symbiotic using the `transferPendingAssets` function. If the withdrawals requested in the current period are insufficient to cover the requested amount, the function will first send all current pending assets and then send pending assets requested in the previous period. Under these conditions, the corner case of sending pending assets to yourself is not handled correctly.

```
if (nextSharesToClaim != 0) {
    toData.sharesToClaim[nextEpoch] += nextSharesToClaim;
    delete fromData.sharesToClaim[nextEpoch];
    amount -= nextPending;
}
```

Specifically, if all the "most recent" pending assets are consumed, the self-transfer will lead to a negative delta due to the way the accounting is done. The pending assets are first added to the sender and then all assets are deleted from the sender. If the sender is the same as the receiver, the full balance (including the received assets) is deleted. This results in a negative delta in the accounting of the pending assets, even though the balance should not change as a result of the transfer.

---

**Code corrected:**

Self-transfers return early and thus bypass the accounting issue.

## 6.22 Insufficient Validation in Migrator

**Informational** **Version 7** **Code Corrected**

*CS-MLW-MULTI-040*

The Migrator reverts when staging if

```
!Migrator(simpleLrtFactory).isEntity(vault)
    && IMellowSymbioticVault(vault).compatTotalSupply() != 0
```

However, it could be meaningful to only allow for successful staging if

- the vault is an entity of the factory
- or the vault is an entity of the previous migrator and has a `compatTotalSupply` of zero.

Ultimately, the validation of vaults could be more restrictive.

---

**Code corrected:**

The code has been corrected to only allow for staging under the conditions described above.

## 6.23 Gas Optimizations

**Informational** **Version 3** **Code Corrected**

*CS-MLW-MULTI-031*

This issue outlines some possibilities for optimizing gas consumption in cases where some noteable gas savings could be made. Note that the list of gas savings is not complete.

In `EigenLayerWithdrawalQueue.transferPendingAssets`, the following iteration is performed:

```
for (uint256 i = 0; i < pendingWithdrawals; i++) {
    ...
    if (accountAssets <= amount) {
        ...
        amount -= accountAssets;
    } else {
        ...
        return;
    }
}
```

Essentially, the function iterates over a set of withdrawals and aims to transfers shares as long as `assets` is not filled.

In case `accountAssets == amount` holds, `amount` will become 0, leading to an exit in the next loop iteration. Thus, the `if` statement could be adjusted to `accountAssets < amount` to exit earlier. That could save *at least* 5 SLOAD operations.

---

**Code corrected:**

In Version 4 the function returns early once `amount` is zero.

## 6.24 Incomplete Deletion of Shares

Informational Version 3 Code Corrected

*CS-MLW-MULTI-032*

`EigenLayerWithdrawalQueue._handleWithdrawal` does the following operation when a user withdraws all shares:

```
if (accountShares == shares) {
    delete _withdrawals[withdrawalIndex];
    accountData_.claimableAssets += assets;
}
```

Note that this does not delete `sharesOf` of the account. As a consequence, the sum of `sharesOf` might exceed the total `shares`.

Note that this issue is amplified by Double tracking of withdrawals and transferred withdrawals due to DoS occurances happening more easily.

---

**Code corrected:**

In Version 4 the `sharesOf` an account are deleted when a withdrawal is handled.

## 6.25 Inconsistent Event Argument Amount in Event Transfer

Informational Version 3 Code Corrected

*CS-MLW-MULTI-019*

The function `SymbioticWithdrawalQueue.transferPendingAssets(0)` emits the event *Transfer* with the argument *amount* when transferring pending assets to the user. The *amount* argument is inconsistent:

1. When transferring pending assets from the next epoch, *amount* is denominated in shares.
2. When transferring pending assets from the current and the next epoch, the event is first emitted with a token amount and then with a shares amount.

This inconsistency can make it difficult for external systems to accurately track the amount of assets transferred.

---

**Code corrected:**

In Version 4 the code has been updated to denominate the *amount* argument in the *Transfer* event is consistently in shares.

## 6.26 Incorrect Argument Order for Key Computation

Informational Version 3 Code Corrected

*CS-MLW-MULTI-033*

The `IsolatedEigenLayerVaultFactory.key` defines the following argument order `owner, strategy, operator`. However, in `IsolatedEigenLayerVaultFactory.getOrCreate` the following order of arguments is provided when using the function: `owner, operator, strategy`.

---

**Code corrected:**

In Version 4, function `getOrCreate` has been updated to use the argument order `owner, strategy, operator`.

## 6.27 Return Value Not Used

Informational Version 3 Specification Changed

*CS-MLW-MULTI-034*

The return value of `EigenLayerWithdrawalQueue._pull` is never used.

---

**Specification changed:**

The return value of _pull has been removed.

## 6.28 Unreachable Adapter Code

Informational Version 3 Code Corrected

*CS-MLW-MULTI-035*

`EigenLayerAdapter.claimWithdrawal` is a function implemented in the adapter allowing to call the isolated vault's respective function. However, the function is unreacheable given that the vault does not delegatecall into the function.

---

**Code corrected:**

The function `EigenLayerAdapter.claimWithdrawal` has been removed in **Version 4**.

## 6.29 Giving Potentially Unnecessary Approvals

**Informational** **Version 1** **Code Corrected**

*CS-MLW-MULTI-018*

Before using the adapter for deposits, approvals are given to the respective subvault. However, such approvals are integration specific.

For example, assume that a protocol does not pull funds but expects some balance delta when the function is called. In such cases, the approval will be unnecessary. Nevertheless, it will be given.

Governance should ensure that only protocols are added that consume the full token approval.

---

**Code corrected:**

The code has been adjusted. Now, adapters must handle approvals.

## 6.30 Implementation Contract Can Be Initialized

**Informational** **Version 1** **Code Corrected**

*CS-MLW-MULTI-008*

The `MultiVault` contracts are deployed as a proxies contract behind a shared implementation. As such, the implementation should not hold any state. However, the `MultiVault` does not disable the initializer function in the constructor allowing anyone to initialize the implementation contract. Writing to the implementation contract has a low impact as it just allows writing to dirty state. Note that the admin in the `MultiVault` can set modules that it delegatecalls into. On chains with selfdestruct (i.e. Avalanche), an attacker could set a module with selfdestruct and subsequently selfdestruct the implementation contract.

---

**Code corrected:**

The constructor of `MultiVault` calls now `_disableInitializers()` which disables calling the initializer on the implementation.

## 6.31 Lack of Events

**Informational** **Version 1** **Code Corrected**

*CS-MLW-MULTI-009*

Events can help front ends and users in retrieving information about the protocol. Below is non-exhaustive list of potentially missing events:

1. `SymbioticWithdrawalQueue.transferPendingAssets()`: Does not emit a relevant event.
2. `RatioStrategy.setRatio()` does not emit an event.
3. In Version 3 the contracts related to EigenLayer have been added to the scope. No events are emitted.

---

**Code corrected:**

Mellow Finance now emits events to signal state changes. Note that for 3. the queue emits respective events. Given that the adapter and the isolated vaults are helper contract it can be reasonable to not emit events there.

## 6.32 Misleading Names of Functions and Variables

**Informational** **Version 1** **Code Corrected**

*CS-MLW-MULTI-010*

Several functions and variables have misleading names:

1. The functions `MultiVault.maxWithdraw` and `ISubVault.maxWithdraw` do not return the maximum amount that can be withdrawn, as their names suggest (similar to `ERC4626.maxWithdraw`). Instead, they return the total position in the corresponding subvault.
2. The function `RatiosStrategy.calculateWithdrawalAmounts` denotes the amount of asset to be requested as `staked`, whereas the function `RatiosStrategy.calculateRebalanceAmounts` calls it `request`. Other inconsistencies in the naming are present in `DepositData`, `WithdrawalData` and `RebalanceData`.

---

**Code corrected:**

1. Corrected. `MultiVault.maxWithdraw` have been renamed to `assetsOf` and `ISubVault.maxWithdraw` to `stakedAt`.
2. Corrected. `request` has been renamed to `staked`.

# 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

## 7.1 EigenLayer Maximum Deposit Considerations

Informational | Version 3 | Acknowledged

*CS-MLW-MULTI-030*

While the `EigenLayerAdapter.maxDeposit` function considers the main cases of limitations of deposits, the following considerations can be made:

1. EigenLayer limits the maximum number of shares by `MAX_TOTAL_SHARES` which is not considered (however, unlikely to be reached).
2. A failing staticcall to `getTVLLimits` might be due to an out-of-gas exception or other reasons. In such cases, the TVL limit would be ignored. However, that is unlikely and of lower impact.

---

**Acknowledged:**

Mellow Finance is aware of the issue and acknowledges it.

# 8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

## 8.1 Frontrunning Initializers

Note Version 1

Initializers can be frontrun if the initialization does not happen in the transaction of the deployment of the proxy. Thus, the deployer should be aware that they should pass the initialization data during proxy construction.

Note that this also holds for upgrades.

## 8.2 Limit Considerations

Note Version 1

Governance and users should be aware that Mellow has local limits. Below are some considerations:

- Note that the vaults implement a limit on the managed assets. Note that the limit can be trivially violated by reducing it below the managed assets or by donating large sums to the vault.
- Further, note that this limit does not consider assets managed by the withdrawal queue owned by users as these are not managed by the Multivault.

## 8.3 Migration Considerations

Note Version 1

Note that it is expected that the migration from the initial version to the simple LRT has been fully completed. If not, issues can arise (the same ones as the previous ones).

Any future upgrade must keep in mind that some storage slots may remain dirty from previous versions. Governance and developers should be aware of this. If not, an issue could arise.

Also, in previous versions of the Vault, withdrawals could be claimed directly from the Vault contract. The new version requires withdrawals to be processed through the `SymbioticWithdrawalQueue`. Integrators must be able to call this queue directly.

Last, migration must follow a strict process. Namely, assets held in Symbiotic by previous versions of the vault will not be recognized by `MultiVault` until Symbiotic is added as a subvault after initialization. This will result in incorrect share calculations. Similarly, that is the case for Symbiotic's default collateral. In case the default collateral is different from the set one, the valuation will be different. However, Mellow Finance is aware of this and will ensure that the contracts' functionalities are paused during the time of migration.

---

**Follow-up:**

As of version 7, a migrator contract is provided.

Migration is only possible if the first migration has been completed (excluding approvals). Additionally, the subvault is added immediately while the default collateral remains the same.

Ultimately, dirty storage slots may remain and approvals might not be fully migrated.

## 8.4 No Information About Rewards Shared With Farms

**Note** **Version 1**

Note that no information about historic or current balances is shared with the distribution farms. Thus, there is no simple and fair possibility of sharing rewards based on on-chain computations.

## 8.5 Non-zero Symbiotic Queue Transfers Can Lead to Zero Transfer

**Note** **Version 1**

The `SymbioticWithdrawalQueue.transferPendingAssets()` allows specifying an underlying asset amount to transfer. However, shares are transferred. Due to rounding down in the divisions (favoring the multivault in integrations), it could be possible that for low input amounts zero shares are transferred (implying zero value transfers). Hence, the transfers might be transferring less value than the input amount specified.

## 8.6 Reentrancy in MultiVault

**Note** **Version 1**

The project is not expected to be used with reentrant tokens. However, several reentrancy issues have been identified if reentrant tokens are used the function `_withdraw()` burns all token shares and then sends claimable assets of each subvault when claiming directly from the withdrawal queue. If tokens have callbacks (e.g., ERC777), an attacker could re-enter after the first transfer. Although `withdraw` has a reentrancy guard, a deprecated exchange rate can be read (read-only reentrancy).

Further, adapters could give the control over the execution to an attacker (e.g. could be part of the external protocols design). Such adapters are not supported. Namely, in such cases the valuation could be wrong. In case of rebalancing operations, the multivault could be drained due to a lack of reentrancy locks. In the case of deposits and withdrawals, there is a possibility of read-only reentrancy similar to the scenarios described above.

## 8.7 Restrictive Pausing Checks for EigenLayer

**Note** **Version 3**

The `EigenLayerAdapter.areWithdrawalsPaused` returns `true` if

```
manager.paused(PAUSED_ENTER_WITHDRAWAL_QUEUE)
    || manager.paused(PAUSED_EXIT_WITHDRAWAL_QUEUE)
    || IPausable(strategy).paused(PAUSED_WITHDRAWALS);
```

Note that this could be too restrictive. Namely, assume a strategy's withdrawals were to be paused. Then, the strategies would assume that any form of withdrawing from EigenLayer would be paused and the funds from other subvaults would be used. However, requests could still be created but are not. Ultimately, the `areWithdrawalsPaused` lacks the capability to distinguish such scenarios.

## 8.8 Strategies Do Not Steer the TVL Down to the Deposit Limit

**Note** **Version 1**

Note that the deposit limit is a soft limit. Namely, it only limits deposit amounts.

However, the strategies could consider the multivaults deposit limit, too. Namely, if the deposit limit had been reduced, the strategies could consider that and try to keep more liquid balances available so that less yield per share is generated. As a consequence, that could incentivize users to exit the system so that the limit is enforced.

## 8.9 Temporary DoS Possibilities for EigenLayer Queue

**Note** **Version 1**

The `EigenLayerWithdrawalQueue` restricts the number of pending withdrawals to `MAX_WITHDRAWALS`. In case of many requests for a given address, that may block that address from requesting. The following consequences are possible:

1. Users could be temporary DoSed from withdrawing if they have many withdrawals pending. However, they can bypass this by specifying another address they own as the recipient.
2. The MultiVault could have its rebalancing DoSed in case too many rebalances had occured that had led to withdrawals. As a result, the rebalancer should be aware that rebalancing should not be carried out too often as otherwise future rebalances could be temporarily DoSed.

Ultimately, a temporary DoS is possible for users and rebalancing actions.

## 8.10 Unsupported Limits in Integrations

**Note** **Version 1**

The specification was adapted to resolve issue Insufficient Limitations for Strategies. Namely, protocols with the following (common) features are not supported.

- Withdrawal upper bound
- Deposit lower bound
- Withdrawal lower bound

See the resolved issue for more details. Note that the above list is not exhaustive and only refers to common limit limitations.

Further, note that in some cases, such as ERC-4626, not all versions implement the functionality (e.g. no withdrawal bounds are common across ERC-4626 tokens). Hence, even though some protocols and standards implement such logic, governance can carefully choose a suitable protocols.