PUBLIC

Code Assessment

of the Mellow LRT Smart Contracts

August 12, 2024

Produced for

Mellow Protocol

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Mellow Team,

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 Mellow LRT according to Scope to support you in forming an opinion on their security risks.

Mellow Finance implements Mellow LRT, a series of modularized smart contracts (a vault with peripheral contracts) to enable permissionless creation of LRTs (Liquid Restaking Tokens) with customized underlying tokens, strategies, and curators.

The most critical subjects covered in our audit are asset solvency, reentrancy, functional correctness and access control. Security regarding functional correctness can be improved, see Unchecked Delegatecall Return Value, while security regarding other subjects is high.

The general subjects covered are precision of arithmetic operation and integration with 3rd party protocols. The reported issues Incorrect rounding direction of ratiosX96Value and Oracle Rounds Down the Price Twice have been resolved, so the security regarding precision of arithmetic operation is satisfactory in the latest version of the codebase. Integration with Lido does not function in line with Mellow Finance's expectation (see Staking Module Deposits May Not Be Allocated to Obol).

The protocol was deployed before this report was finalized and some low severity findings remain unresolved, see Findings. Accounts with privileged roles should be aware of the potential risks and actively monitor the state of the vault.

In addition, special care should be taken by the vault's admins and operators to avoid front-running (see Front-running of Important Updates) and reentrancy (see Tokens with Transfer Hooks Enable Reentrancies).

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

Furthermore, we have verified the deployment of the contracts, please refer to the deployment verification files for more information about the contracts at respective addresses.

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	0
Medium-Severity Findings	2
• Code Partially Corrected	1
• Acknowledged	1
Low-Severity Findings	17
• Code Corrected	2
• Specification Changed	1
• Risk Accepted	6
• Acknowledged	8

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 Mellow LRT 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	05 Jun 2024	e31044584c781c7c9960df7b1c311e24c8b7d92a	Initial Version
2	08 Jul 2024	1c885ad9a2964ca88ad3e59c3a7411fc0059aa34	Version with Fixes
3	02 Aug 2024	79361bae88fa93580f88be9ef998e9794db2ba17	Final Version

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

The following files were in the scope of this review:

```
src/Vault.sol
src/VaultConfigurator.sol
src/modules/DefaultModule.sol
src/modules/erc20/ERC20SwapModule.sol
src/modules/erc20/ERC20TvlModule.sol
src/modules/erc20/ManagedTvlModule.sol
src/modules/obol/StakingModule.sol
src/modules/symbiotic/DefaultBondModule.sol
src/modules/symbiotic/DefaultBondTvlModule.sol
src/oracles/ChainlinkOracle.sol
src/oracles/ConstantAggregatorV3.sol
src/oracles/ManagedRatiosOracle.sol
src/oracles/WStethRatiosAggregatorV3.sol
src/security/AdminProxy.sol
src/security/DefaultProxyImplementation.sol
src/security/Initializer.sol
src/strategies/DefaultBondStrategy.sol
src/strategies/SimpleDVTStakingStrategy.sol
src/utils/DefaultAccessControl.sol
src/utils/DepositWrapper.sol
src/utils/RestrictingKeeper.sol
src/validators/AllowAllValidator.sol
src/validators/DefaultBondValidator.sol
src/validators/ERC20SwapValidator.sol
src/validators/ManagedValidator.sol
```

Version 3 is used for the deployment and all the contracts (in scope) are the same as **Version 2**.

2.1.1 Excluded from scope

Any file not mentioned explicitly above, such as tests, scripts, configurations, and other dependencies, were not in the scope of this review. Furthermore, third-party protocols such as Symbiotic, Lido and Chainlink oracles, that Mellow LRT interacts with are excluded from this review. Tokens to be used as underlying assets in Vaults are also excluded from the scope of this review.

Finally, the soundness of the financial model was not evaluated.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

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 offers Mellow LRT, a series of modularized smart contracts (a vault with peripheral contracts), which enables permissionless creation of Liquid Restaking Tokens (LRT) with customized underlying tokens, strategies, and curators.

A Mellow LRT is an ERC-20 compliant vault (deployed with a configurator, oracles, strategies and modules), where users can mint shares proportional to the valuation of deposited tokens. Each LRT features different curators (admins and operators), who have the privilege to allocate funds to different strategies (e.g. delegation to different AVS: Actively Validated Services).

2.2.1 Vault

The Vault is the core of a Mellow LRT. A vault configurator will be deployed in the constructor of the vault, which stores important parameters and implements the logic for updating them. Each vault is configured with a set of underlying tokens that can change over time.

2.2.1.1 Deposit

Users mint shares by depositing the underlying tokens when calling the function `deposit()`. Tokens approvals should be granted to the vault in advance. The following operations are performed on deposits:

1. The deadline and configurator's deposit lock are checked and the call is validated by the validator contract. The vault allows to restrict access to `deposit()` only to whitelisted addresses.
2. The underlying tokens can only be deposited in a balanced way according to the `ratiosOracle`.
3. The shares minted to the recipient (`to`) are proportional to the total value deposited over the vault underlying token's TVL (total value locked). The TVL is reported by a set of customizable TVL modules and the price are denoted in a base token.
4. The first deposit can only be made by the admin or operators, and the lp amount are checked with the `minLpAmount` (slippage protection) and the total supply should respect the `maximalTotalSupply`.
5. After minting new shares to the recipient, a `depositCallback()` will be made if the callback address specified by configurator is non-zero.

After user's deposit, the admins and operators have the privilege to trigger external calls and delegatecalls from the vault to allocate the deposit tokens into third party protocols (e.g. strategies, bonds):

- `externalCall()` - triggers a low-level `call` and returns its status and response. The calling contract and the selector passed in the low-level `call` should be whitelisted.

- `delegateCall()` - it is checked that the delegatecall is made to an approved delegate module, and validated by a validator contract.

2.2.1.2 Withdraw

As tokens may have been deposited into third party protocols, instant withdrawals from the vault may not be satisfied, hence the withdrawal logic is implemented in three phases:

- `registerWithdrawal()` - withdrawals should first be registered by the users, which specifies a recipient address `to`, the `lpAmount` to be burnt, and an array of underlying token's `minAmount` for slippage protection.

    1. The deadline of this transaction and the withdraw request are validated first.
    2. Each user can register at most one withdrawal at any time, hence a flag `closePrevious` can be submitted to cancel an existing withdrawal request of the user.
    3. The `lpAmount` cannot be 0, and will be automatically capped by the balance of the user.
    4. A withdrawal request struct will be stored, which snapshots the hash of the current underlying tokens.
    5. User's share will be transferred to the vault temporarily.

- `processWithdrawals()` - withdrawals can be process by the admin or operators before expiry (`requestDeadline`). An array of users can be specified and processed in a batch:

    1. The withdrawal context will be prepared in `calculateStack()`.
    2. Each withdrawal request will be analyzed (`analyzeRequest()`) regarding the cached context. The withdraw shall respect the withdrawal ratio defined in the `ratiosOracle`.
    3. In case the request has expired, the underlying tokens have been changed, or the slippage protection is violated, the processing is deemed impossible and cancelled.
    4. In case the withdrawal is possible but the vault's current balance is lower than the expected amount, the withdrawal is currently impossible and skipped.
    5. For possible withdrawals, tokens will be attributed to the recipient, and the withdrawal request will be deleted. The total `lpAmount` of successful withdrawals will be burnt at the end. Withdrawals are subject to a fee, which remains in the vault.
    6. At the end, a `withdrawalCallback()` will be made if the callback address specified by configurator is non-zero.

- `emergencyWithdraw()` - a force withdrawal can be trigger by the user if the withdrawal request has not been processed after a `emergencyWithdrawalDelay` before its expiry. This functionality serves as a last resort for LPs to exit from the vault in case curators of the vault are not reachable, hence they do not process withdrawals on time.

    1. The transaction deadline, withdrawal expiry and `emergencyWithdrawalDelay` are checked first. In case the withdrawal has expired, it will cancel the withdrawal and return an empty array of `actualAmounts` instead of reverting.
    2. The withdrawal will be made as the base tokens the vault holds, the `actualAmounts` are proportional to the `lpAmount` over the total shares.
    3. The `actualAmounts` are compared with `minAmounts` for slippage protection.
    4. At the end, the withdrawal request will be deleted and the `lpAmount` will be burnt.

Users can cancel their withdrawals anytime via `cancelWithdrawalRequest()`, which simply deletes the request struct and their address from the `_pendingWithdrawers`, and transfers the lp shares back to the user.

2.2.1.3 Other Functions

`receive()` has been implemented to receive Ether when unwrapping WETH.

The admins have the privilege to the following functions:

- `addToken()` - add a new underlying token, which will be arranged in an ascending order.
- `removeToken()` - remove an underlying token, whose underlying amounts reported by the TVL modules must be 0.
- `addTvlModule()` - add a non-repeatable TVL module, its reported TVL data must only contain underlying tokens.
- `removeTvlModule()` - remove an existing TVL module. Admins are responsible to add a new TVL module in the same transaction and make sure that the valuation of the vault is always correct.

The Vault is ERC-20 compliant hence implements the standard ERC-20 interfaces. The internal function `_update()` has been overridden to apply the potential transfer lock between users, while minting / burning shares are not blocked.

The following view functions are further provided:

- `withdrawalRequest()` - returns the withdrawal request of a given user.
- `pendingWithdrawersCount()` - returns the pending withdrawal requests length.
- `pendingWithdrawers(uint256 limit,uint256 offset)` - returns an array of addresses with pending withdrawal requests, starting from a specified offset with a limited length.
- `pendingWithdrawers()` - returns all the pending withdrawers.
- `underlyingTokens()` - returns an array of underlying tokens.
- `isUnderlyingToken()` - returns whether a token is a member of underlying token.
- `tvlModules()` - returns an array of TVL modules' addresses.
- `underlyingTvl()` - returns the underlying TVL reported by the TVL modules (arrays of tokens addresses and amounts).
- `baseTvl()` - returns the base TVL reported by the TVL modules.

The Vault also inherits all the interfaces of `DefaultAccessControl`.

2.2.2 Default Access Control

`DefaultAccessControl` inherits `AccessControlEnumerable` which offers the role-based access control with enumerating functionalities. In the constructor of `DefaultAccessControl`, an address is passed and set as `ADMIN_ROLE` and `OPERATOR`. The `ADMIN_ROLE` is the admin of the `ADMIN_ROLE` and `ADMIN_DELEGATE_ROLE`. And the `ADMIN_DELEGATE_ROLE` is the admin of `OPERATOR`.

- `isAdmin()` - returns if an address has `ADMIN_ROLE` or `ADMIN_DELEGATE_ROLE`.
- `isOperator()` - returns if an address has `OPERATOR` role.
- `requireAdmin()` - requires if an address has `ADMIN_ROLE` or `ADMIN_DELEGATE_ROLE`.
- `requireAtLeastOperator()` - requires if an address has `ADMIN_ROLE`, `ADMIN_DELEGATE_ROLE` or `OPERATOR` role.

2.2.3 Vault Configurator

Vault Configurator registers important parameters for a vault, and is managed by the vault admins and operators. It implements a two-phase update logic via stage and commit for most parameters:

- An update to a parameter should be staged first. The timestamp of this operation is stored.

- A staged update can only be committed after a predefined delay.
- A staged update can also be deleted (rollback).

A `_baseDelay` will be enforced for the updates of itself and other delays. In addition, revoke functionality has been added for deposit lock and delegate module approval exclusively:

- `revokeDelegateModuleApproval()` - the approval of a delegate module can be removed immediately by the admin without any delay.
- `revokeDepositsLock()` - the deposit lock can be removed immediately by at least the operator immediately without any delay.

2.2.4 Vault Proxy

The vault is intended to be deployed behind a Transparent Upgradeable Proxy contract. Upon the deployment of the vault proxy, a `ProxyAdmin` contract will be deployed as an admin of the vault, which has the privilege to upgrade the implementation via `upgradeAndCall()`. The `owner` of the `ProxyAdmin` has the ultimate power to upgrade the vault implementation.

The `owner` of the `ProxyAdmin` is critical for the security of vaults, hence it should be carefully protected. Mellow Finance has implemented the contract `AdminProxy` which can serve as the `owner` of `ProxyAdmin`. However deployment scripts in (Version 3) do not use this setup, see Owner of ProxyAdmin is a multisig.

`AdminProxy` has 3 roles: a proposer, an acceptor, and an emergency operator. The acceptor can remove or grant roles to any address. This contract stores a base implementation as a safe fallback. New proposal of implementation or base implementation can be made by either the proposer or the acceptor. Only valid proposals with a correct implementation and valid `calldata` should be accepted. The acceptor can update the new base implementation by accepting it (acceptor can even propose and accept a new implementation in the same TX). It can also accept a queued proposal, which triggers a `upgradeAndCall()` on the `ProxyAdmin` to upgrade the vault implementation.

A `latestAcceptedNonce` has been used to avoid accepting an old proposal. The acceptor can reject all the proposals by setting the `latestAcceptedNonce` to the length of proposal queue.

The emergency operator has the privilege to immediately reset the implementation to the base implementation (`resetToBaseImplementation()`). This action can be performed only once by the emergency operator.

A `DefaultProxyImplementation` is provided as a base implementation in emergency, which is an ERC-20 with transfer locks. Hence the allowance is the only state that can be changed by the users and no transfers can be made.

As the vault features no initialization functionality for the proxy, an `Initializer()` has been provided as the initial implementation to initialize the `DefaultAccessControl` and `ERC-20` name and symbol for the vault.

2.2.5 Oracles

A wrapper contract around Chainlink oracle has been implemented.

- `setBaseToken()` - the admin of each vault has the privilege to set the the vault's base token.
- `setChainlinkOracles()` - the admin of each vault has the privilege to set the `AggregatorData` of corresponding tokens. Admin should ensure that oracles for the underlying token and the base token have the same quote asset.
- `getPrice()` - returns the last answer and decimals fetched from the aggregator, the reported timestamp is subject to a freshness check against the `maxAge`.
- `aggregatorsData()` - returns the data of a specific token in a vault.
- `priceX96()` - returns the price of a token denominated in a vault's base token in the X96 format.

The following contracts are further implemented:

- `ConstantAggregatorV3` returns a constant answer with 18 decimals.
- `WStethRatiosAggregatorV3` will fetch the onchain conversion rate between wstETH and stETH with 18 decimals.
- `ManagedRatiosOracle` maintains the deposit and withdraw ratios of vault's underlying tokens.

    1. It can only be updated by the vault's admin via `updateRatios()`. The ratios should have the same length as vault's underlying tokens, and sum up to `2^96`.
    2. The ratios of a vault can be retrieved from `getTargetRatiosX96()`, where the vault's underlying tokens are validated and should match the hash of underlying tokens by the time the ratios were set.

2.2.6 Modules

Modules for ERC-20 tokens, Symbiotic restaking protocol, and Obol staking module on Lido are implemented to be used by the vault. TVL modules should be configured carefully to not account assets redundantly.

2.2.6.1 ERC-20 Modules

- `ERC20TvlModule` - function `tvl()` will retrieve the ERC20 balance of a vault's underlying tokens. This module can only be called via external calls.
- `ManagedTvlModule` - vault's admin can manually set the tvl data of the vault via `setParams()`, which requires the data must be underlying tokens. The data could be retrieved later via the getter `tvl()`. This module can only be called via external calls.
- `ERC20SwapModule` - this module implements the general swap logic for a vault. `swap()` should be called via a delegatecall.

    1. it will increase the approval to the `to` address and initiate a low-level call for the swap.
    2. the difference of the pre- and post-balance of token-in and token-out are checked for slippage protection.
    3. in case the approval is not fully consumed, it will reset the approval to 0 at the end.

2.2.6.2 Symbiotic Modules

A `DefaultBondModule` is provided to handle the deposit and withdrawal from Symbiotic Bonds (Default Collateral contracts), which can only be triggered via delegatecalls.

- `deposit()` - it will increase the allowance for the bond contract, and invest into the bond via `bond.deposit()`;
- `withdraw()` - it will cap the withdraw amount to its ERC-20 balance and invoke `bond.withdraw()`;

A `DefaulBondTvlModule` is also provided, which does not accept delegatecalls. where the admin of a vault can set an array of bonds' addresses via `setParams()`, whose `asset()` must be one of vault's underlying tokens. The getter `tvl()` will query the balance of bond tokens and always assume a 1:1 conversion rate between the bond token and its underlying token.

2.2.6.3 Obol Staking Module

A helper contract `StakingModule` has been created to facilitate depositing funds into Lido and creating new validators for the Obol staking module. It can only be called with delegatecalls.

- `convert()` - it withdraws ETH from WETH, deposit into Lido, and wrap it into wstETH.

- `convertAndDeposit()` - this function deposits idle WETH of the vault to Lido and trigger the process to create new validators for Obol Staking Modules:

    1. It reverts if Lido's unfinalized withdrawal is greater than the buffered ETH.
    2. It estimates the new validators allocated to the Obol `stakingModuleId` with the available ETH and cap the deposited WETH by the amount allocated to Obol.
    3. Eventually it will convert WETH to wstETH and trigger the creation of new validators via `depositBufferedEther()`, which requires valid guardian signatures above quorum from the Lido's Deposit Security Committee.

2.2.7 Strategies

Two strategies are built on top of the modules: `DefaultBondStrategy` and `SimpleDVTStakingStrategy`.

2.2.7.1 Default Bond Strategy

This strategy is bounded to a vault, a `ERC20TvlModule` and a `bondModule`. It features the `DefaultAccessControl` where the admin can set the deposit ratios into different bonds (`setData()`) and the ratios should sum up to `2^96`.

`depositCallback()` can be invoked by the vault or at least the operator of this strategy. It triggers `delegateCall()` from the vault, which splits and deposits the idle underlying tokens of the vault into the bonds according to the ratios.

Two withdraw functions can be invoked by at least the operator. `processAll()` will try to process all the pending withdrawers and `processWithdrawals()` will only process the specified input users:

1. `delegateCall()` will be triggered from the the vault to withdraw all the underlying tokens from the bonds.
2. `vault.processWithdrawals()` will be triggered to analyze the withdraw request and process the possible withdrawals.
3. The remaining underlying tokens will be deposited back to the bonds.

2.2.7.2 Simple DVT Staking Strategy

This strategy is bounded to a vault and a `stakingModule`. It also features the `DefaultAccessControl` where the admin can set the `maxAllowedRemainder`.

`convertAndDeposit()` is a permissionless function, which triggers a delegatecall from the vault to the stakingModule to deposit into Lido and create new validators for Obol.

`processWithdrawals()` can only be called by at least the operator of this strategy. An input `amountForStake` can be passed to trigger a delegatecall from the vault to the stakingModule, in order to convert WETH to wstETH before the withdrawal (`vault.processWithdrawals`). The wstETH amount cannot exceed `maxAllowedRemainder` after withdrawal.

2.2.8 Validators

Validators are used by the vault to inspect the caller, callee, and calldata:

2.2.8.1 Allow All Validator

It does not conduct any checks.

2.2.8.2 Default Bond Validator

It features the `DefaultAccessControl`, where the admin can set the supported bond addresses. In `validate()`, it requires 68 bytes calldata, which can only be a call to `deposit()` or `withdraw()` to the supported bond.

2.2.8.3 ERC20 Swap Validator

It features the `DefaultAccessControl`, where the admin can set the supported routers (`setSupportedRouter()`) and tokens (`setSupportedToken()`). In `validate()`, it requires at least 292 bytes of calldata, which must be a call to `swap()`. And the router, token-in and token-out must be supported. It will also revert if:

1. token-in equals token-out.
2. `amountIn` or `minAmountOut` is 0.
3. the deadline has passed.
4. `swapData` is less than 4 bytes.

2.2.8.4 Managed Validator

The managed validator features customized access control. It uses a bitmask to store users' roles. Each role may call:

1. all functions on specified contracts.
2. restricted selectors on specified contracts.

Roles can be granted and revoked by authorized calls. In case a roles is set as a public role, this role is accessible by all users.

In addition, customized validators can be added for a target address as an extension to the current permission check: upon calling `validate()`, it will check the permission first(`requirePermission()`), then invoke the extended validation logic from the `customValidator` if it exists.

2.2.9 Roles and Trust Model

Liquid Restaking Tokens (Vaults) can be created and managed by different curators with different strategies. Each vault has a set of privileged accounts that control and manage funds deposited into a vault and should be trusted to behave always correctly and be non-malicious. Users should be fully aware of the potential risks of different Vaults before depositing into them.

The vault is assumed to be deployed as a Transparent Upgradeable Proxy, hence the `owner` of `ProxyAdmin` is fully trusted, otherwise the vault can be upgraded to a malicious implementation and get drained. If the `owner` of `ProxyAdmin` is set to the contract `AdminProxy`, the following assumptions should hold:

- The acceptor is fully trusted, otherwise it can upgrade the vault to a malicious implementation and drain all assets.
- The proposer is not trusted, and the acceptor should reject malicious implementations proposed.
- The emergency operator is semi-trusted. It can at most upgrade the vault to the base implementation once, which will temporarily DoS the system in case of `DefaultProxyImplementation`.

In addition, the vault features `DefaultAccessControl`, where all three roles: admin, admin delegate, and operator are fully trusted. Otherwise, they may trigger whitelisted external or delegatecalls from the vault for their own interests (depending on validators configuration). The vault's roles further oversee the following contracts: vault configurator, ratios oracle, chainlink oracle, managed tvl module, and default bond tvl module. In case the admin is compromised, attacks could be but not restricted to the following scenarios:

- The configuration could be manipulated in the interest of the roles (i.e. set a higher withdraw fee, trigger DoS of the system).
- The deposit/withdraw ratios of different tokens can be manipulated.
- The underlying tokens can be changed frequently to block the withdrawal from being processed.
- The oracle can be set to a malicious contract and get manipulated.
- The tvl in the managed tvl module could be set to any desired value.
- decrease the tvl reported by removing bonds from the default bond tvl module, or double accounting the same bond.

For some configurations of the vault, the admin should update them in an atomical way, otherwise, the updates can be sandwiched to arbitrage the system, for instance: should a TVL module be replaced, the removal of old module and addition of new module should be finished atomically.

The vault's operator is also a powerful role and is fully trusted, otherwise it can trigger swaps with bad parameters (slippage, swap path) to drain the vault.

Managed Validator's admin is fully trusted to not misbehave and collude with the vault's roles, otherwise they can whitelist and trigger external or delegate calls to malicious contracts.

The default bond strategy's roles are fully trusted to set the deposit ratios of different bonds correctly. They should process the users' withdrawal requests on time, and in the worst case they are inactive, users can still do emergency withdrawals.

The simple DVT staking strategy's roles are also fully trusted, if they are inactive, users can still do emergency withdrawals.

The bonds used (at the time of this review) is `DefaultCollateral` of Symbiotic. These contracts are fully trusted to work as expected (i.e. there is no slippage on minting and withdrawal, and the conversion rate is always 1:1).

The `Lido.depositBufferedEther` is expected to work correctly as documented. Curators of a vault should continuously monitor and take measures if Lido's oracle is inactive or reports incorrect data. Obol staking module and Lido's Deposit Security Committee are assumed to be trusted. Similarly, Chainlink oracles are considered trusted.

The underlying tokens used by the vault are expected to be legitimate ERC-20 tokens without special behaviors (i.e. inflationary/deflationary, rebasing, transfer hooks, transfer fees, etc.).

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 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	2

- Staking Module Deposits May Not Be Allocated to Obol Acknowledged
- Unchecked Delegatecall Return Value Code Partially Corrected Acknowledged

Low-Severity Findings	14

- Emergency Withdrawal Amounts May Be Over-Estimated Acknowledged
- Emergency Withdrawal Will Cancel Expired Request Silently Acknowledged
- Ether Could Be Stuck in Deposit Wrapper Risk Accepted
- Front-running of Important Updates Risk Accepted
- Inaccuracies Regarding WStETH Price Freshness Acknowledged
- Inconsistent Access Controls in Strategies' processWithdrawals Acknowledged
- Incorrect Addresses in Constants Acknowledged
- Initialization of Implementations Acknowledged
- Missing Sanity Checks Acknowledged
- Possible to Block Token Removal Risk Accepted
- Price Zero Accepted by ChainlinkOracle Risk Accepted
- ProcessAll Is Subject to Gas Griefing Attacks Risk Accepted
- Unused Debt Flag in TVL Modules Acknowledged
- Vault's First Deposit Restrictions Risk Accepted

5.1 Staking Module Deposits May Not Be Allocated to Obol

Design Medium Version 1 Acknowledged

CS-MELLOWLRT-001

In `StakingModule`, `convertAndDeposit()` implements the logic for the vault to:

1. deposit into Lido (convert WETH to wstETH).

2. trigger the creation of new validators for the Obol staking module on Lido.

It first estimates the number of validators for a staking module `maxDepositsCount` that can be created with the idle funds on Lido (buffered Ether minus unfinalized withdrawal) and the WETH balance of the vault. Then it caps the amount of Ether to deposit by the `maxDepositsCount * 32 ether`. And it finally invokes the `depositBufferedEther()` on Lido's deposit security module to trigger the allocation and creation of validators for a specific `stakingModuleId` of Obol.

However, Lido implements a min-first allocation algorithm, which will always allocate the idle Ether across all active staking modules. Hence the vault's deposit may be all, partially, or not allocated to the Obol staking module.

Even though a specific `stakingModuleId` is specified in this call, it only triggers the creation of new validators to this module id if there is any allocation. This behavior is not in line with Mellow Finance's expectation.

Acknowledged:

Mellow Finance is aware that the staked Ether is not always allocated to Obol staking module. The possible scenarios are described in Staking Amount May Be Used in Different Ways.

5.2 Unchecked Delegatecall Return Value

Security | Medium | Version 1 | Code Partially Corrected | Acknowledged

CS-MELLOWLRT-002

In vault, `delegateCall()` does not check the return value of the opcode `delegatecall`. Instead, it forwards the success flag and response as return values.

However, the strategies do not verify the return value of `delegateCall()`:

In `DefaultBondStrategy`:

- The `depositCallback()` triggers a `vault.delegateCall()`, using the bond module to deposit into the bonds according to designated ratios. Hence if any `deposit()` call failed, the `depositCallback()` will still succeed and the resulted allocation may actually violates the ratios.
- The `_processWithdrawals()` triggers a `vault.delegateCall()`, using the bond module to withdraw from all the bonds. In case any `withdraw()` call failed, only funds withdrawn from other bonds will be used to finalize the withdrawal request. This may also leads to unexpected allocation across all bonds.

In `SimpleDVTStakingStrategy`:

- `processWithdrawals()` does not check the forwarded return value of `vault.delegateCall()` (used to deposit `amountForStake` into Lido). As a result, the actual staked amount may be inconsistent from the value reflected in the events.

Acknowledged:

Code has been partially corrected: the return value of `delegateCall()` is now checked in the function `SimpleDVTStakingStrategy.convertAndDeposit()`. Furthermore, the contract `DefaultBondModule` has been revised to avoid reverting when functions `depositCallback()` and `_processWithdrawals()` are called.

However, the return value remains unchecked in `SimpleDVTStakingStrategy.processWithdrawals()`. Mellow Finance acknowledged the issue but has decided to keep the relevant code unchanged.

5.3 Emergency Withdrawal Amounts May Be Over-Estimated

Design Low Version 1 Acknowledged

CS-MELLOWLRT-019

Emergency withdrawal enables the withdrawal of tokens reported by the tvl modules (`baseTvl()`), which returns an array of tokens and the respective amounts and debts the vault has for each token.

```
(address[] memory tokens, uint256[] memory amounts) = baseTvl();
```

However, the tokens to withdraw are computed based on the current ERC-20 balance that the vault holds, instead of the amounts reported by the tvl module.

```
uint256 amount = FullMath.mulDiv(
    IERC20(tokens[i]).balanceOf(address(this)),
    request.lpAmount,
    totalSupply
);
```

As a result, the withdrawable amount may be over-estimated (without considering the debt), hence the user may receive more tokens. This leads to solvency issues for the vault.

Acknowledged:

Mellow Finance acknowledged the issue and provided the following reasoning:

```
The logic by which emergencyWithdraw will be called implies that the system is already
working incorrectly (due to the inaccessibility of admin/operator wallets or due to
some other reasons). Formally, here we are no longer looking at withdrawalFeeD9 or
debt tokens, but simply want to give a proportional portion of the tokens to all users.
```

We would like to emphasize that emergency withdrawals might happen also due to gas griefing attacks, see ProcessAll Is Subject to Gas Griefing Attacks.

5.4 Emergency Withdrawal Will Cancel Expired Request Silently

Design Low Version 1 Acknowledged

CS-MELLOWLRT-003

In case a withdrawal request is not processed before its deadline, the request expires and cannot be finalized anymore. In case an `emergencyWithdraw()` call is invoked with this request, the expired withdrawal will be silently cancelled and the function returns an empty array of amounts.

From the return value, it is unable for the caller to distinguish the following cases:

1. a successful emergency withdrawal with 0 tokens obtained (in case `minAmounts` is also an array of zero).
2. an expired emergency withdrawal which gets cancelled.

Acknowledged:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

5.5 Ether Could Be Stuck in Deposit Wrapper

Design | Low | Version 1 | Risk Accepted

CS-MELLOWLRT-004

A deposit wrapper has been provided to facilitate depositing Ether, WETH, stETH, and wstETH into the vault with only wstETH as the underlying token. Users should specify the token address to deposit, and in case the token address is 0, it will be regarded as a deposit of Ether attached to this call (msg.value).

However, in case a non-zero `msg.value` is attached to the call, with the token address as WETH, stETH, or wstETH, it will ignore the attached Ether and only deposit the ERC-20 token.

Risk accepted:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

5.6 Front-running of Important Updates

Security | Low | Version 1 | Risk Accepted

CS-MELLOWLRT-005

The vault relies on the admins to update the configurations properly, and is dependent on the price reported by the oracles. Hence the following front-running scenarios are possible:

- The admin's update to managed TVL module can be sandwiched by users to arbitrage.
- The Chainlink price updates and Lido oracle reports can be sandwiched to mint or redeem shares in favor of the user.

The probability of such attacks is relatively low given that users cannot withdraw immediately unless two conditions hold:

1. The update transaction that changes valuation of the vault is pending.
2. Operator has submitted a transaction to process all pending withdrawals in a vault.

Risk accepted:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

5.7 Inaccuracies Regarding WStETH Price Freshness

Correctness | Low | Version 1 | Acknowledged

CS-MELLOWLRT-006

The function `WStethRatiosAggregatorV3.latestRoundData` queries `WStETH` to get the price and always sets the variables `updateAt` and `answerAt` to `block.timestamp`:

```
function latestRoundData() public view override
    returns (uint80, int256, uint256, uint256, uint80)
{
    return (0, getAnswer(), block.timestamp, block.timestamp, 0);
}
```

Note that the returned values for `answerAt` and `updateAt` do not correctly reflect the freshness of `WStETH` price, which depends on the Lido oracle. Hence, if Lido oracle fails to update the `WStETH` price frequently, the oracle `WStethRatiosAggregatorV3` provides incorrect information about the price freshness.

Acknowledged:

Mellow Finance is aware of the inconsistency regarding the freshness of the reported price. The code remains unchanges as Mellow Finance considers Lido's oracle to be updated always regularly.

5.8 Inconsistent Access Controls in Strategies' processWithdrawals

Correctness | Low | Version 1 | Acknowledged

CS-MELLOWLRT-007

In `DefaultBondStrategy`, `processWithdrawals()` requires the `msg.sender` to be at least the operator of this strategy, otherwise the call will revert.

However in `SimpleDVTStakingStrategy`, `processWithdrawals()` checks the length of users first, which will return early if the input length is 0. Only after the length check, the role of the `msg.sender` is checked. As a result, a call to this function can succeed even though the `msg.sender` is not authorized (although no state changes are made in this case).

Acknowledged:

Mellow Finance is aware of this issue but has decided to keep the code unchanged. Third-party integrators should be aware of this behavior.

5.9 Incorrect Addresses in Constants

Correctness | Low | Version 1 | Acknowledged

CS-MELLOWLRT-008

The library `Constants` declare multiple constant variables that are used for testing against Ethereum mainnet chain. However, the following variables appear to store wrong values:

1. WSTETH_CHAINLINK_ORACLE stores the Chainlink price feed for rETH instead of wstETH.
2. WETH_CHAINLINK_ORACLE stores the Chainlink price feed for steth instead of WETH.
3. SIMPLE_DVT_MODULE_ID stores id 1 instead of 2.

Acknowledged:

Mellow Finance is aware of these inconsistencies but these values are used only for testing.

5.10 Initialization of Implementations

Security Low Version 1 Acknowledged

CS-MELLOWLRT-010

Mellow vaults are deployed behind a proxy scheme. The first implementation of vault should be contract Initializer, and later it is set to the contract Vault. We would like to highlight that both implementations have a role setup that potentially are controlled by untrusted accounts. However, these accounts can only manipulate the storage of the implementation contract, not the storage of the proxy.

The admin of the Vault implementation is set on constructor(). This account can create new roles and call privileged functions such as delegateCall(), therefore potentially triggering a call to SELFDESTRUCT. However, since Cancun upgrade, the semantics of SELFDESTRUCT have changed and it does not destroy the contract anymore (unless called in same transaction with contract creation).

Similarly, the function initialize() in the contract Initializer can be called by anyone to set the name, symbol and admin of the implementation contract to arbitrary values.

Acknowledged:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

5.11 Missing Sanity Checks

Design Low Version 1 Acknowledged

CS-MELLOWLRT-011

Several functions in the codebase can implement sanity checks to avoid mistakes and misconfigurations. We provide a non-exhaustive list of such cases:

1. The function AdminProxy.proposeBaseImplementation() checks that implementation is non-zero, however it does not check that the proposed address has actually code and can serve as implementation. If the implementation has no code due to a misconfiguration, the function resetToBaseImplementation() triggered by the emergency operator would revert.
2. Functions upgrade**() in AdminProxy do not perform any sanity check on the address of the new account. In case of upgradeAcceptor(), accidentally setting this role to addr(0) locks the contract.
3. The function SimpleDVTStakingStrategy.setMaxAllowedRemainder() does not perform any sanity check on newMaxAllowedRemainder.
4. The function ManagedValidator.grantPublicRole() does not check that role is not 255 which would allow anyone call admin functions, thus taking over the validator contract.

Ackowledged:

Mellow Finance acknowledges the missing checks but has decided to keep the code unchanged.

5.12 Possible to Block Token Removal

Design | Low | Version 1 | Risk Accepted

CS-MELLOWLRT-013

Function `Vault.removeToken()` permits the removal of an underlying token only when its total value locked (TVL) is zero:

```
function removeToken(address token) external nonReentrant {
    ...
    (address[] memory tokens, uint256[] memory amounts) = underlyingTvl();
    ...
    for (...) {
        if (tokens[i] == token) {
            if (amounts[i] != 0) revert NonZeroValue();
            ...
}
```

One can make such operations to fail by frontrunning the transaction calling `removeToken()` and donate `1 Wei` of the token to the vault, hence forcing the victim transaction to revert with error `NonZeroValue`.

Risk accepted:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

5.13 Price Zero Accepted by ChainlinkOracle

Correctness | Low | Version 1 | Risk Accepted

CS-MELLOWLRT-014

The internal function `ChainlinkOracle._validateAndGetPrice()` reverts if the retrieved price is negative or it is considered stale:

```
if (signedAnswer < 0) revert InvalidOracleData();
answer = uint256(signedAnswer);
if (block.timestamp - data.maxAge > lastTimestamp) revert StaleOracle();
```

However, if the Chainlink aggregator returns an invalid price of `0`, it is still accepted by ChainlinkOracle.

Risk accepted:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

5.14 ProcessAll Is Subject to Gas Griefing Attacks

Security Low Version 1 Risk Accepted

CS-MELLOWLRT-015

The function `processAll()` tries to process all existing withdraw requests in the queue. Hence, it can run out of gas in case there are too many withdrawals, or malicious users front-run this to register a lot of dusty withdrawals. The operators should select to only process the desired requests in such cases. However, operators should eventually process dust withdrawal requests to avoid errors during emergency withdrawals, see Emergency withdrawal amounts may be over-estimated. Moreover, operators do not have clear incentives to pay gas costs for all withdrawal requests, see Missing Incentives for Operators.

Risk accepted:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

5.15 Unused Debt Flag in TVL Modules

Design Low Version 1 Acknowledged

CS-MELLOWLRT-017

The struct `ITvlModule.Data` is declared as follows:

```
struct Data {
    address token; // Address of the token
    address underlyingToken; // Address of the underlying token
    uint256 amount; // Current amount of the token
    uint256 underlyingAmount; // Current amount of the underlying token
    bool isDebt; // Flag indicating if the token represents debt
}
```

TVL modules in the codebase do not use the flag `isDebt`, hence it is always set to the default value `false`. However, the function `Vault._calculateTvl()` implements additional for-loops to account for cases when `isDebt` is set to `true`.

Acknowledged:

Mellow Finance is aware of this behavior but has decided to keep the code unchanged.

5.16 Vault's First Deposit Restrictions

Security Low Version 1 Risk Accepted

CS-MELLOWLRT-018

The internal function `_processLpAmount()` restricts the first deposit in a vault only to privileged accounts, i.e., only accounts with roles either `operator` or `admin` can perform the first deposit:

```
if (totalSupply == 0) {
    // scenario for initial deposit
    _requireAtLeastOperator();
```

```
    lpAmount = minLpAmount;
    if (lpAmount == 0) revert ValueZero();
    if (to != address(this)) revert Forbidden();
}
```

The function does not restrict the minimum tokens that should be locked in the first deposit. Furthermore, no restrictions are set on the `lpAmount` that are minted to the vault itself. Although the first depositor is trusted, misconfigurations can still happen. For example, choosing a large `lpAmount` (close to maximum `uint`) can render the vault useless, while choosing a low `lpAmount` impacts the rounding error for user deposits.

Moreover, the initial LP shares minted to the vault can be recovered by privileged accounts via functionalities `externalCall()` or `delegateCall()`, hence there is no guarantee that inflation attacks are ruled out after the first deposit.

Risk accepted:

Mellow Finance is aware of this issue but has decided to keep the code unchanged.

6 Resolved Findings

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

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	0
Low-Severity Findings	3

- Incorrect Rounding Direction of ratiosX96Value Code Corrected
- Oracle Rounds Down the Price Twice Code Corrected
- Specification Mismatch for MAX_ORACLE_AGE Specification Changed

6.1 Incorrect Rounding Direction of ratiosX96Value

Security Low Version 1 Code Corrected

CS-MELLOWLRT-009

In vault, `calculateStack()` will prepare the context for processing withdrawals, which computes the average price of tokens according to the withdrawal ratios.

```
for (uint256 i = 0; i < tokens.length; i++) {
    uint256 priceX96 = priceOracle.priceX96(address(this), tokens[i]);
    s.totalValue += FullMath.mulDiv(amounts[i], priceX96, Q96);
    s.ratiosX96Value += FullMath.mulDiv(s.ratiosX96[i], priceX96, Q96);
    s.erc20Balances[i] = IERC20(tokens[i]).balanceOf(address(this));
}
```

The average price `s.ratiosX96Value` is rounded down, and the rounding errors will accumulate in the loop in case there are multiple underlying tokens. Users may get a lower price and hence withdraw more tokens (`expectedAmounts`) back.

The impact of the incorrect rounding may be limited due to:

- The `expectedAmounts` computed in `analyzeRequest()` is rounded down in 3 occurrences.
- In case there is a withdrawal fee applied, the `expectedAmounts` will be reduced.

Note that the rounding errors should be in favor of the Vault to avoid solvency issues.

Code corrected:

The rounding direction has been changed to round up, hence `s.ratiosX96Value` is not under-valued anymore.

6.2 Oracle Rounds Down the Price Twice

Correctness **Low** **Version 1** **Code Corrected**

CS-MELLOWLRT-012

To compute the price of a token denominated in a base token, the oracle fetches the respective prices and divide them with the oracle decimals and token decimals into consideration. However, it rounds down twice in its computation, which removes accuracy from the price returned if both the oracle and token use 18 decimals.

```
priceX96_ = FullMath.mulDiv(
    tokenPrice,
    Q96,
    10 ** (decimals + IERC20Metadata(token).decimals())
);
priceX96_ = FullMath.mulDiv(
    priceX96_,
    10 ** (baseDecimals + IERC20Metadata(baseToken).decimals()),
    baseTokenPrice
);
```

Code corrected:

Code has been corrected to include all the computations into one `FullMath.mulDiv()` hence it will only round down once.

6.3 Specification Mismatch for MAX_ORACLE_AGE

Correctness **Low** **Version 1** **Specification Changed**

CS-MELLOWLRT-016

The documentation of the contract `ChainlinkOracle` specifies a maximum age for the prices returned by the oracle:

```
MAX_ORACLE_AGE: The maximum allowable age for Chainlink oracle data, set to 2 days.
```

However, the contract does not enforce a maximum age for oracle data. The vault `admin` can freely set a threshold for each price feed.

Specification changed:

Mellow Finance acknowledged the inconsistency between code and the Notion documentation, and consider the code to comply to the intended behavior.

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 Approve Is Not Paused in the DefaultProxyImplementation

Informational Version 1

CS-MELLOWLRT-020

In case the vault's implementation is set to the `DefaultProxyImplementation`, most functions will be paused but users can still query the balance and metadata of the Vault token (LRT). No transfers can be made in this case, however, `approve()` is not paused and users can still modify their allowances.

7.2 Computation of baseTVL in emergencyWithdraw May Be Unnecessary

Informational Version 1 Acknowledged

CS-MELLOWLRT-024

The public function `Vault.baseTVL()` may consumes non-trivial gas to retrieve the tokens and respective amounts from the tvl modules. It is called internally only from `emergencyWithdraw()` which only checks if TVL for a token is zero:

```
(address[] memory tokens, uint256[] memory amounts) = baseTvl();
...
for (uint256 i = 0; i < tokens.length; i++) {
    if (amounts[i] == 0) {
        if (minAmounts[i] != 0) revert InsufficientAmount();
        continue;
    }
    ...
}
```

However, the reported amounts are never used in the computation of withdrawable amounts (which is instead computed based on the vault's balance of the ERC-20 token `balanceOf()`).

7.3 Gas Optimizations

Informational Version 1 Acknowledged

CS-MELLOWLRT-021

The following parts of the codebase can be optimized for gas efficiency:

1. Multiple contracts of the system use mappings in the format: `mapping(key_type => bool)`. Solidity uses a word (256 bits) for each stored value and performs some additional operations when operating `bool` values (due to masking). Therefore, using `uint` instead of `bool` in mappings `isSupportedBond`, `isSupportedRouter`, `isSupportedToken` is slightly more efficient.

2. The local variable `isUnderlying` in function `Vault.isUnderlyingToken()` is not used.
3. When deploying a Vault, a new `VaultConfigurator` contract is deployed and linked only by the implementation contract. Therefore, the deployment of `VaultConfigurator` could be avoided to save gas.
4. The internal function `Vault._processLpAmount()` checks if address `to` is the zero address, which is redundant with the check in `ERC20._mint()`.
5. The wstETH ratios oracle fetches the conversion rate between stETH and wstETH from the wstETH contract, it could be more gas efficient by directly querying the stETH `getPooledEthByShares()`.
6. The struct `IVaultConfigurator.Data` uses 3 storage slots for `value`, `stagedValue` and `stageTimestamp`. However, given that the largest values need 160 bits (type `address`), it is possible to optimize the struct such that `stagedValue` and `stageTimestamp` are packed in a single storage slot.
7. The interface `IChainlinkOracle` declares the error `Forbidden` which remains unused.
8. The interface `IVault` imports the library `Arrays` which remains unused.
9. The contract `ManagedRatiosOracle` can optimize the storage use by storing in the mapping `vaultToData` an array of `Data` instead of performing encoding which does not pack the values.
10. Similarly, the contract `ManagedTvlModule` can optimize the storage use by storing an array of `Data` in the mapping `vaultParams` and reorder the variables in the struct `Data` to benefit from packing.
11. The first check of `(amount == 0)` in function `DefaultBondModule.deposit()` is redundant when called from `DefaultBondStrategy._deposit()`.

7.4 Possible to Mark Variables as Constants

Informational **Version 1** **Acknowledged**

CS-MELLOWLRT-022

Several contracts in the codebase store the address of 3rd party tokens such as `WETH`, `stETH`, or `wstETH`. These addresses are typically provided on contract deployment, e.g., in `DepositWrapper`, `WStethRatiosAggregatorV3` and `StakingModule`. However, such variables could be declared as constants to avoid potential misconfigurations during deployment.

7.5 Staking Amount May Be Used in Different Ways

Informational **Version 1** **Acknowledged**

CS-MELLOWLRT-025

The vault estimates the `maxDepositsCount` of Obol with the idle funds on Lido (buffered Ether minus unfinalized withdrawal) and the WETH balance of the vault. Then it caps the amount to deposit by the `maxDepositsCount * 32 ether`. In the following cases, the deposited amount will be used in different ways (assume the deposit cap on Lido will never be reached):

- Assume Lido has 30 depositable ether, mellow has 2 WETH, and all ether will be allocated to Obol:
    1. `maxDepositsCount` will be 1.
    2. amount to deposit will be 2 ether, which will be all used for Obol.

- Assume Lido has 32 depositable ether, mellow has 2 WETH, and all ether will be allocated to Obol:
    1. `maxDepositsCount` will be 1.
    2. amount to deposit will be 2 ether, which becomes idle funds on Lido.
- Assume Lido has 0 depositable ether, mellow has 64 WETH, and 32 ether will be allocated to other staking module first, then the rest 32 are allocated to Obol:
    1. `maxDepositsCount` will be 1.
    2. amount to deposit will be 32 ether, which will be all used for other staking module.

Acknowledged:

Client has acknowledged the different scenarios and stated that from these cases only the first one is intended.

7.6 Vault Accepts Ether Transfers

Informational | Version 1 | Acknowledged

CS-MELLOWLRT-023

The function `receive()` is implemented in the vault to receive Ether transfers, this is useful to unwrap WETH or handle ether withdrawal from other protocols. As `receive()` does not restrict the caller (`msg.sender`), accidental Ether transfers to the vault are possible and are treated as donations.

7.7 convertAndDeposit Will Revert if Lido Unfinalized Withdrawal Is Greater Than Buffered Ether

Informational | Version 1 | Acknowledged

CS-MELLOWLRT-026

In case the stETH for unfinalized withdrawals is greater than the buffered ether on Lido, `convertAndDeposit()` will always revert to avoid depositing vault's WETH to Lido and creating new Obol validators.

```
if (bufferedEther < unfinalizedStETH)
    revert InvalidWithdrawalQueueState();
```

In case the amount of unfinalizedStETH is relatively small and WETH held by the vault is enough to cover both the unfinalized withdrawals and creation of new validators, in theory it is still possible to create new Obol validators, though part of the vault's deposit will be used to cover the withdrawals first.

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 Admins and Operators Should Monitor Large Emergency Withdrawals

Note Version 1

Emergency withdrawals are only expected to happen when the system is not working correctly. As the emergency withdrawal benefits from no withdrawal fees and no specific withdrawal ratios, the admins and operators should monitor the vault to avoid irregular large emergency withdrawals and process them in advance. Otherwise, the system may become insolvent.

8.2 Capping of Large Values

Note Version 1

The function `Vault.registerWithdrawal()` allows users to pass a large value for `lpAmount` to be withdrawn, however it is capped by the user's balance:

```
address sender = msg.sender;
...
uint256 balance = balanceOf(sender);
if (lpAmount > balance) lpAmount = balance;
```

Similarly, the function `DefaultBondModule.withdraw()` limits the amount of bond tokens withdrawn to the balance of the vault:

```
uint256 balance = IDefaultBond(bond).balanceOf(address(this));
if (balance < amount) amount = balance;
...
IDefaultBond(bond).withdraw(address(this), amount);
```

3rd-party integrator should be aware of these specifics and handle correctly cases when users provide large values that are then passed to the functions above.

8.3 Careful Selection of Emergency Delay

Note Version 1

The emergency delay should be selected carefully. In case it is too short, users have higher chance to trigger it before the operators can process the withdrawal requests. The emergency withdrawal has the following advantages for the users:

1. It does not respect the withdrawal ratios defined by the oracle, hence an unbalanced withdraw may be possible.
2. It does not incur any withdraw fees.

8.4 Emergency Withdrawals Depend on Withdrawal Deadlines

Note **Version 1**

Upon registering a withdrawal, users set a request deadline which limits the time window for a regular withdrawal or an emergency withdrawal. In case the chosen deadline is sooner than the emergency withdrawal delay, the emergency withdrawal can be triggered. Users should be aware of this and set the withdrawal deadline properly for their requests.

8.5 Escalation of Privileges in ManagedValidator

Note **Version 1**

The contract `ManagedValidator` has an `admin` role that has the ultimate control over the access control configurations. The role `admin` can create other roles that might be restricted on calling specific contracts, or only some functions in a target contract. However, there is a possibility of privilege escalation for roles that can call functions `grantPublicRole()` or `grantRole()` in the contract `ManagedValidator` itself. Such roles can assign themselves the `admin` role and take over the contract.

The account holding `admin` role is responsible for configuring the access control correctly and carefully assess permissions of other accounts.

8.6 External Call Return Value Should Be Checked

Note **Version 1**

The function `Vault.externalCall()` does not check the return value of the low level call, and simply forwards the success flag with the response. The caller of `externalCall()` should carefully check these values to revert in case of unexpected execution result.

8.7 Incompatible ERC20 Tokens

Note **Version 1**

The underlying assets in a vault should be ERC20-compliant. However, tokens that exhibit special behaviors do not integrate well with vaults and should not be used. In addition to the tokens already mentioned in Roles and Trust Model, we would like to highlight some the following functionalities that might break compatibility with vaults if used as underlying assets:

- Tokens with blocklists: withdrawal functionalities in a vault rely on the assumptions that transfers always succeed. However, for some tokens this assumption does not hold always. For instance, tokens that implement blocklist revert on transfers related to specific addresses. This functionality can be misused by attackers to request withdrawals for accounts that are currently blocklisted in an underlying token, hence the withdrawal cannot be processed. If the account is removed from the blocklist in the future, the emergency withdrawal can be performed which may be unexpected for the vault.

- Pausable tokens: if one of the underlying tokens is paused, then the main functionalities of the vault (such as depositing, withdrawing, processing requests) cannot be performed.
- Special transfer amounts: tokens that cap the amount transferred in `transferFrom()` to user's balance should not be used as underlying assets. Otherwise, attackers can mint vaults shares for free.

8.8 Missing Incentives for Operators

Note Version 1

The admins or operators are important roles to maintain the vault, update configurations correctly, and process users withdrawals. However, there is no explicit incentive for them to complete these tasks (especially withdrawals for users) given that the transaction gas costs are non-negligible.

8.9 Operator's Privilege on Swap Module

Note Version 1

The vault's operator is a privileged role. In case operators become malicious, they may be able to trigger certain calls from the vault for their own interests when using the swap module. The swap validator does not validate the data used for calling the swap router hence:

1. The operator can set a bad slippage parameters for the vault when using the swap module, which can be sandwiched by their own transactions to drain the vault's tokens.
2. The operator can deploy a malicious token with a callback to themselves exclusively, and inject it to the swap path. Then they can reenter to call vault `deposit()`, where they can mint underpriced shares due to token-in has been transferred out for the swap but token-out has not been received yet.

8.10 Owner of ProxyAdmin Is a Multisig

Note Version 3

The deployment script in Version 3 sets the `owner` of `ProxyAdmin` to a multisig instead of the contract `AdminProxy`. The vault implementation is therefore changed only when a transaction with enough signatures is submitted. Differently from `AdminProxy`, we would like to emphasize that the multisig offers no easy way to change the implementation to the default implementation in an emergency unless the required signatures are collected.

8.11 Process Withdrawal Should Be Atomical

Note Version 1

Function `processWithdrawals()` is implemented in the strategies to prepare the tokens before processing the withdrawal request in the vault. In case the operators or admins handle withdrawals directly in the vault, then the withdrawing of funds from the strategies (preparing idle funds) and triggering `Vault.processWithdrawals()` should be done atomically in the same transaction. Otherwise, one can call permissionless functionalities such as `convertAndDeposit` in `SimpleDVTStakingStrategy` or trigger callbacks that deposit the idle funds back to the strategies, hence leaving the vault with no tokens needed for the withdrawal requests.

8.12 Removal of TVL Module

Note Version 1

When removing a TVL module, no checks are performed. This operation should be handled carefully by the privileged accounts with the admin role. Removing a TVL module that reports non-zero assets has direct impacts in the accounting of the vault and can create arbitrage opportunities to mint underpriced LP shares. Therefore, the account triggering `removeTvlModule()` should ensure that a new TVL module is added to report TVL for the respective assets, and no operation from untrusted users can happen in between.

8.13 Simple DVT Staking Strategy Withdraw Ratio

Note Version 1

In case a withdrawal request is processed, it will withdraw the underlying assets according to a specific ratio. For a vault that uses the strategy `SimpleDVTStakingStrategy`, WETH will be deposited into Lido and the vault gets wstETH in return, however, no logic has been implemented to withdraw ETH from Lido and wrap it again into WETH. Hence the withdrawal ratio should be set with a high weight on wstETH, otherwise the vault may not have enough WETH to cover the withdrawal and only emergency withdrawal is possible.

8.14 Tokens With Transfer Hooks Enable Reentrancies

Note Version 1

We assume that tokens with transfer hooks are not used as underlying assets for a vault (see Roles and Trust Model), otherwise reentrancy vulnerabilities are enabled and the consequences are severe.

For instance, the contract `ChainlinkOracle` implements a function `setBaseToken()` that restricts access to the `admin` of a vault. If operations from `admin` can be initiated by anyone as long as valid signatures are provided (e.g., Gnosis Safe wallet), the following frontrunning attack is possible:

1. Current base token for a vault is `WETH`.
2. `admin` submits a `TX1` that calls `setBaseToken()` to set `DAI` as base token.
3. Attacker frontruns the valid ``` `` ```TX1 that changes the base token, and performs the following:

    3.1 Call `Vault.deposit()`. 3.2 On transfer hooks, reenter in `ChainLinkOracle` and submit `TX1` which changes the base token. 3.3 Accounting in `Vault.deposit()` is broken as the deposit value and total value of remaining tokens are denominated in `DAI` instead of `WETH`.

In addition, 3rd party protocols should be aware that read-only reentrancy is possible during the token transfer callbacks, where the state updates are not completed. For instance, the price per share (vault's TVL divided by the shares total supply) could be manipulated.

8.15 Users Should Revoke Unused Allowance for the Vault

Note **Version 1**

Upon depositing into the vault user will pass an array of token amounts, and `deposit()` will only use the amounts proportional to a deposit ratio. Hence if users approve more than the actually deposited amount, there could be remaining allowance, which in theory could be transferred out by malicious vault admins by `vault.externalCall` or `vault.delegateCall`.

Users should only approve the actual deposited amount, and revoke the remaining allowance due to changes from the deposit ratios.

8.16 Vault Shares Should Not Be Used as Underlying Tokens by the Vault

Note **Version 1**

The vault shares should not be used as an underlying token of the vault itself, which will break the internal accounting.

In addition, the shares held by the vault (for pending withdraw requests) should not be swapped, or transferred to other addresses by the operators. Otherwise, process withdrawals will revert due to insufficient shares for burning, which will trigger a DoS attack.