PUBLIC

Code Assessment

of the Mellow Vaults Smart Contracts

Aug 09, 2022

Produced for

by

Contents

1 Executive Summary

Dear Mellow team,

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

Mellow Finance implements an investment protocol that pools investors funds and manages these funds according to an investment strategy smart contract.

We value the very good and professional communication with the Mellow team and the quick response time. In the initial review and the following iterations, we uncovered an unusual number of issues with many high severity issues. Many of these issues would have been caught by proper testing. The code base appeared to be not ready for the review when the review started. After multiple iterations the code base was improved significantly.

All raised issues were addressed and most were fixed by code changes. We conclude that the reviewed contracts currently provide a satisfactory level of security. But our experience shows that a high amount of discovered issues has an increased tail risk of more undiscovered issues. Additionally, not all code was in scope of the review (see 2.1 Scope). Hence, we want to highlight that security reviews complement but don't replace other vital measures to secure a project, like e.g., limited testing phases or bug bounties. We did our best to help to eliminate all severe issues, but it is important to note that security audits are time-boxed and cannot uncover all vulnerabilities.

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	5
• Code Corrected	5
High-Severity Findings	14
• Code Corrected	14
Medium-Severity Findings	17
• Code Corrected	14
• Specification Changed	3
Low-Severity Findings	53
• Code Corrected	37
• Specification Changed	4
• Code Partially Corrected	2
• Acknowledged	10

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 Vaults 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	25 Feb 2022	f79ea5fc82b8ae5e0fa488c5aae023e893edf7f0	Initial Version
2	18 Apr 2022	736b34243ced0ea0346dd874438354db6dafa135	Version 2
3	23 May 2022	b4250761505742c211428986ecb4189ae2e402fc	Version 3
4	08 Jun 2022	3c73391bb182e910b3d714dab1930d4e885dfccf	Version 4
5	15 Jun 2022	5e09b7372f02d8f81f6d66d827d5883323b86a03	Version 5
6	20 Jun 2022	bb66a637c4a5834c4dace90205776159726b8299	Version 6
7	11 Jul 2022	1fae4225ccd6164ad613e0860619ecd10287dd39	Version 7
8	29 Jul 2022	ff8f3ba89c362663ef9f3a0ff31d7bbf95457629	Version 8
9	03 Aug 2022	ed3e07e5b873dbe6f4e5d632d0adc1f5b47dec8e	Version 9

For the solidity smart contracts, the compiler version `0.8.9` was chosen.

2.1.1 Excluded from scope

The following files were out of scope:

- contracts/ContractRegistry.sol
- contracts/libraries/SemverLibrary.sol
- contracts/oracles/UniV2Oracle.sol
- contracts/utils/BatchCall.sol
- contracts/utils/ContractMeta.sol
- contracts/utils/DefaultProxy.sol
- contracts/utils/DefaultProxyAdmin.sol
- contracts/libraries/external/*

Additionally, the `MellowVault` and the functions to compute the square root in `CommonLibrary.sol`, were put out of scope while performing the code assessment.

2.1.2 Excluded from this report

During the engagement shortcomings inside the smart contracts were discovered by the Mellow team. These shortcomings are not listed in this report.

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 an investment protocol that pools investors funds and manages these funds according to an investment strategy smart contract.

The overall system has certain parameters managed by The `ProtocolGovernance` smart contract. Different vaults are responsible to keep the funds and/or invest them in other DeFi protocols like AAVE, YEARN or Uniswap. A root vault is the overarching connector for all vaults. The root vault is the entry point for a user to invest funds. Strategy contracts balance the ratios of tokens held in the vaults and between the vaults.

A user who wants to invest funds will send the funds to the root vault. The root vault will in return issue a corresponding amount of liquidity provider tokens to track the user's investment to the user. The funds will end up in a special vault which acts as a cash position. As soon as a strategy manager invokes the vault rebalancing in the connected strategy, the strategy will distribute the funds from the cash vault to the investment/integration vaults. These vaults will use the funds to invest into the third party DeFi protocols like Aave. When a user decides to redeem/withdraw their liquidity provider tokens for the corresponding share of tokens, the root vault will drain the cash vault and if needed take more money from the investment/integration vaults.

The system has multiple roles that need to be trusted fully or partially. These are:

The `DefaultAccessControl` smart contract uses OpenZeppelin's `AccessControlEnumerable` scheme to set up the following roles:

- `ADMIN_ROLE`
- `ADMIN_DELEGATE_ROLE`
- `OPERATOR`

The contract is inherited by `UnitPricesGovernance` and, consequently, `ProtocolGovernance`. The `DefaultAccessControl` is also used in the `ChainlinkOracle`, `UniV3Oracle` and `LStrategy`.

Initially, the deployer of these contracts can set an address that is granted the:

- `OPERATOR` and
- `ADMIN_ROLE` roles.

A slightly modified version of the `DefaultAccessControl` that features a late initialization to work with proxies is used by the `MStrategy`.

Access for vaults is controlled by NFTs that are minted during registration of vaults in the `VaultRegistry`. For example, when deploying the `YearnVault` by calling the `createVault` function in `YearnVaultGovernance`, the contract is registered at the `VaultRegistry`, where an NFT is minted to an owner address that can be specified when creating the vault.

When creating a root vault, the NFTs of the sub-vault are transferred to the root vault to give the root vault the required control over the sub-vaults. The root vault's NFT itself stays under the ownership of the address given by the deployer who can approve another address, giving it `Strategy` access level. Strategies can therefore be managed by different entities than vaults.

The system deployment and setup are quite complicated. The following steps try to outline the setup chronologically.

1. These steps can be performed in any order:
    - Deploy the protocol `ProtocolGovernance` contract.

    - Deploy the oracle contracts.
2. These steps can be performed in any order:
    - Deploy the validator contracts.
    - Deploy the `ContractRegistry` contract.
    - Deploy the `VaultRegistry` contract.
3. Deploy the vault governance contracts.
4. Call `createVault` on the integration/sub-vaults that are needed.
5. In case a `UniV3Vault` is used:
    1. Mint an initial Uniswap v3 liquidity position with a random EOA.
    2. Approve the EOA for spending the NFT received from the `VaultRegistry`.
    3. Transfer the Uniswap v3 LP NFT to the newly created `UniV3Vault`.
6. Deploy a strategy with the reference to the desired vaults (The `MStrategy` contract acts as a factory whereas the `LStrategy` needs to be deployed normally).
7. Create the root vault with the desired integration/sub-vaults and the strategy.

The contracts have the following functionalities:

- `ProtocolGovernance`

  Manages the following variables with a time delayed commit scheme:
    - The protocol's `Params` struct that includes:
        - `uint256 maxTokensPerVault`
        - `uint256 governanceDelay`
        - `address protocolTreasury`
        - `uint256 forceAllowMask`
        - `uint256 withdrawLimit`
    - The permissions for certain addresses (e.g., creating a vault, registering a vault, being a vault token or passing requirements set by the validators)
    - The validator contracts that are called when an external call is done (e.g., swapping tokens on an exchange)

  Revoking/Removing permissions or validators can be done instantly with no time lock.

  Roles: The `admin` role of this contract should be fully trusted as it sets all the critical parameters of the system. In general, the parameters are updated with the stage-delay-commit pattern, so users should continuously monitor the staged parameters for malicious values and react before they are committed. However, for revoking permissions or validators, there is no delay enforced.
- `UnitPricesGovernance` is a contract inherited by the `ProtocolGovernance`. It allows the admin role to set reference prices used to calculate the withdraw limit in `ProtocolGovernance.withdrawLimit`. The admin should carefully set the price values taking into consideration the decimals of the respective token.
- `ContractRegistry` allows the protocol governance admin or operator to register contracts. The contract is not used in the system at all but seems to be called externally to verify if contracts belong to the system.

- `VaultRegistry` manages the vault's NFTs. Each time a vault is created, it will be registered in this contract and receive a unique ID represented by an NFT. The NFT is also used for access control purpose. The relevant roles are the owner and the approved spender of the NFT.

  Roles: The `admin` of the protocol governance has special privileges also in this contract. Besides updating the state parameters, in the first versions of the code the admin could approve any `vault` NFT to any arbitrary address via `adminApprove`, however this functionality was removed in Version 5.

- Validator contracts are used to verify calls done by the `externalCall` function in the Integration vaults. `externalCall` allows every vault nft owner or approved account to perform any low-level call to other contracts as long as a validator contract for that contract has been set in the `ProtocolGovernance` and the corresponding validator successfully verified the calldata.

- Oracles are used by the strategies to balance the portfolio between vaults and tokens. The root vault uses oracles to convert the TVL of integration vaults to the same token in `_getTvlToken0`. The `UniV3Vault` uses the oracle price in `_getMinMaxPrice` to determine TVL.

  Roles: The Chainlink and UniV3Oracle inherit the `DefaultAccessControl`, and the `admin` has the privileges to set the oracle address (in case of Chainlink) or the Uniswap pool.

- Vault governance contracts are factory contracts that can create new vaults. They additionally store some vault type specific information in the `delayedProtocolParams` object (e.g., AAVE lending pool address).

  Roles: These contracts inherit the roles of protocol governance.

- There are two types of integration vaults. The `ERC20Vault`, which is a cash position vault, stores tokens to act as a buffer when tokens are withdrawn. It is worth mentioning that the Yearn vault has a default `maxLoss` parameter set to 100%, basically omitting any protection for the user if the Yearn strategy had made a loss. `ERC20Vault` is also the hub which receives all deposits from the aggregation/root vault. The tokens remain in the `ERC20Vault` until a strategy starts balancing and, hence, distributing the tokens to the connected other integration vaults. The other integration vaults are adapters to other DeFi projects (AAVE, Yearn, Uniswap, Mellow). They invest all tokens they receive and only store the respective project tokens.

- The `ERC20RootVault` is the main entry and exit point for users. The vault acts as an aggregator and oversees all sub-vaults that are connected to it. By calling `withdraw` and `deposit` users can close or open a position. The user's position is represented by liquidity tokens minted to the investing user.

  Roles: The privileged roles in this contract are the `admin` of the protocol governance, the strategy linked to the vault and owner of the vault NFT. End users also can access the functionalities to `deposit` and `withdraw` funds.

2.3 Strategies

Strategy contracts manage the token distribution between and inside vaults. Operators can call certain functions that rebalance tokens between vaults but cannot send tokens outside of the ecosystem.

2.3.1 `MStrategy`

`MStrategy` is a factory contract to create new strategies. Anyone can deploy a clone of the implementation by calling the function `createStrategy` and providing the following parameters: `tokens_`, `erc20Vault` (cash position), `moneyVault_` (investment position), `fee_` and `admin`. `MStrategy` supports only vaults with exactly two tokens, and both `erc20Vault` and `moneyVault_` should have same tokens.

The main functionality of `MStrategy` is `rebalance` which can be called only by the `admin` of the strategy. The first operation of the `rebalance` is to transfer tokens between the `erc20Vault` and the `moneyVault` to maintain a healthy ratio as defined in the parameters `ratioParams`. The ratio is

computed on terms of the total value locked (TVL) by both positions. This is achieved by calling the internal function `_rebalancePools`. The second part of `rebalance` is to maintain the ratio of tokens in the cash and investment position in line with the ratio of these tokens in a Uniswap pool. If the cash position has more from one token than desired, the function `_rebalanceTokens` computes the amount of tokens that need to be swapped in Uniswap, which is performed in `_swapToTarget`.

Roles: The `admin` address is important as it is a privileged address that can set other roles for the contract as described above. Furthermore, this address has access to the following functionalities:

- `rebalance`: moves the funds between the cash position and the investment according to the desired ratio. The function should be called with carefully crafted inputs for slippage protection.
- `manualPull`: can move any amount of funds at any time between the `erc20Vault` and `moneyVault`.
- `setOracleParams`: updates the oracle parameters, such as the number of observations considered for Uniswap oracles, maximum tick deviation supported, and slippage tolerance.
- `setRatioParams`: sets the ratio of funds in the cash position and the investment vault.

2.3.2 LStrategy

The second strategy of the project is `LStrategy`. This strategy is supposed to be deployed once, so it does not provide any cloning functionalities. `LStrategy` operates on three vaults, the first is still the cash position, while the other two are Uniswap positions that are setup in a such a way to cover the current price tick and therefore optimize the earnings.

The Uniswap positions are referred as `lowerVault` - cover the current tick and a predefined number of lower ticks, and `upperVault` - covers the current tick and a predefined number of upper ticks. Both `lowerVault` and `upperVault` overlap with each other and ideally both should always cover the current tick, where most of the activity happens. If the price goes up, i.e., current tick increases, the strategy should move more liquidity to the `upperVault`, otherwise moves liquidity to the `lowerVault`. If one position does not cover anymore the current tick, then all liquidity is moved to the other vault, and a new position in Uniswap is minted. Note that, `LStrategy` should have enough tokens to open new Uniswap positions in order to follow the price as intended. Given that `LStrategy` is not supposed to hold any token, the strategy operator (or anyone) should continuously donate the required tokens to `LStrategy`.

The main functionalities of the strategy are:

- `rebalanceERC20UniV3Vaults`: moves the funds between the cash position (`erc20Vault`) and the two Uniswap positions (`lowerVault` and `upperVault`) according to the desired ratio.
- `rebalanceUniV3Vaults`: this function moves the funds between the `lowerVault` and `upperVault` depending on the price change by calling the `internal` function `_rebalanceUniV3Liquidity`. If one of the Uniswap positions does not cover anymore the current price tick, the function `_swapVaults` is called to close the outdated position and mint a new one.
- `postPreOrder` and `signOrder`: `LStrategy` uses Cowswap for the token exchange and these functions allow privileged accounts to save a pre-order as a state variable and sign the Cowswap order.
- `collectUniFees`: this function calls `collectEarnings` in the Uniswap positions `lowerVault` and `upperVault`.
- `manualPull`: allows the `admin` to transfer any amount of funds at any point from one vault to another one.
- Updating params functions: allow the `admin` to update the `tradingParams`, `ratioParams` and `otherParams` of the strategy.

Roles: The `admin` address is important as it is a privileged address that can set other roles for the contract as described above. Furthermore, this address has access to the following functionalities:

`manualPull` and the functionalities to set the strategy parameters, including the ones for the slippage protection. The `operator` role can trigger the execution of functions `rebalanceERC20UniV3Vaults`, `rebalanceUniV3Vaults`, `postPreOrder`, and `collectUniFees`.

2.3.3 Tokens

We assume that only ERC20-compliant tokens with no callback features are whitelisted and used by the system. The support of tokens with callback hooks, like ERC777 or ERC677, is out of scope.

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	0
Low-Severity Findings	12

5.1 Inconsistent Decimals of LP Token

Correctness Low Version 5 Acknowledged

The function `ERC20RootVault.deposit` performs the following checks when new LP tokens are minted to a user:

```
require(lpAmount + balanceOf[msg.sender] <= params.tokenLimitPerAddress, ExceptionsLibrary.LIMIT_OVERFLOW);
require(lpAmount + totalSupply <= params.tokenLimit, ExceptionsLibrary.LIMIT_OVERFLOW);
```

The LP tokens distributed by root vaults do not have pre-defined number of decimals but depend on the token amounts of the first deposit, hence making difficult to set the params `tokenLimitPerAddress` and `tokenLimit` in advance.

Acknowledged:

Mellow Finance acknowledges the issue and will take care to set the proper limits after initial LP shares are minted and the respective decimals are known:

```
We don't intend to stand limits in advance of the launch of the system, we rather want
to stand them as MaxUint256 initially and then have a possibility to set meaningful values
based on the supply of lp tokens during the work of the system.
```

5.2 Performance Fee in Specific Setups

Design Low Version 5 Acknowledged

The performance fee is charged in `ERC20RootVault` only if the price of LP tokens has increased in value, which is calculated in the statement:

```
uint256 lpPriceD18 = FullMath.mulDiv(tvlToken0, CommonLibrary.D18, baseSupply);
```

However, in specific setups where the `token0` is of high value but has low decimals, while the `token1` is of low value but with many decimals, the variable `baseSupply` would inherit the decimals of `token1`. Therefore, in such setups it is possible that the statement above returns `lpPriceD18` equal to zero.

Acknowledged:

Mellow Finance has decided to keep the code unchanged as they only will use only verified token combinations that this issue does not occur. The response:

```
We decided that this situation would not be possible when calculating the performance fee,
since we agreed to use only verified tokens, for which the difference between decimals
would be less than 18.
```

5.3 Possible Optimization in AggregateVault

Design Low Version 5 Code Partially Corrected

The function `AggregateVault._push` performs the following actions:

1. Approves allowance with `safeIncreaseAllowance` for each token to `destVault`.
2. Calls `destVault.transferAndPush`, which transfers `tokenAmounts` to the ERC20Vault.
3. Resets approval to `destVault` for all tokens to `0`.

Given that the `_push` function moves tokens to the ERC20Vault and allowance in the end should be `0`, the function can be revised to be more efficient. For instance, `safeIncreaseAllowance` performs additional operations and is useful when the existing allowance is not zero and should be considered. Also, the function consumes in step 2 the allowance given earlier, hence the last `for-loop` might be omitted.

Code partially correct:

The function `AggregateVault._push` is made more efficient by performing the external calls `safeIncreaseAllowance` and `safeApprove` only for tokens that non-zero amounts are being

transferred (`tokenAmounts[i] > 0`). However, for the other tokens two external calls are performed for updating the allowance.

5.4 Possible Optimization on Deposits and Withdrawals

Design | Low | Version 5 | Acknowledged

The function `ERC20RootVault.deposit` can be optimized to be more gas efficient by transferring the tokens directly from the user to the ERC20Vault. Currently, the tokens are first transferred from the user to the root vault:

```
for (uint256 i = 0; i < tokens.length; ++i) {
    ...
    IERC20(tokens[i]).safeTransferFrom(msg.sender, address(this), normalizedAmounts[i]);
}
```

and then, in `AggregateVault._push` tokens are transferred again:

```
for (uint256 i = 0; i < _vaultTokens.length; i++) {
    IERC20(_vaultTokens[i]).safeIncreaseAllowance(address(destVault), tokenAmounts[i]);
}
```

Similarly, the function `ERC20RootVault.withdraw` can be made more efficient if the tokens are transferred directly from the sub-vaults to the user instead of transferring to the root vault first and then to the user.

Acknowledged:

Client acknowledges the optimization possibility but prefers to keep the code unchanged:

```
The main idea behind this behavior is for the root vault to be responsible for pushing
tokens onto different vaults. We consider the current design to be clearer with
pushing with the ```AggregateVault._push``` method.
```

5.5 Redundant Calculation of LP Amounts

Design | Low | Version 5 | Acknowledged

The function `ERC20RootVault.deposit` calculates the LP amount that is rewarded to the user two times:

```
{
    ...
    (preLpAmount, isSignificantTvl) = _getLpAmount(maxTvl, tokenAmounts, supply);
    for (uint256 i = 0; i < tokens.length; ++i) {
        normalizedAmounts[i] = _getNormalizedAmount(...);
        ...
    }
}
actualTokenAmounts = _push(normalizedAmounts, vaultOptions);
(uint256 lpAmount, ) = _getLpAmount(maxTvl, actualTokenAmounts, supply);
```

Initially, `preLpAmount` is calculated based on the `tokenAmounts`, then `normalizedAmounts` are computed. Considering that `_push` moves tokens to the ERC20Vault, the returned `actualTokenAmounts` is equal to `normalizedAmounts`. Hence, recomputing `lpAmount` is redundant.

Acknowledged:

Client acknowledges the redundant calculation of LP amount but prefers to keep the code unchanged as in the future the behavior of ERC20Vault might change, i.e., the returned `actualTokenAmounts` might not be equal to `normalizedAmounts`.

5.6 Broad Access Control for Functions

Design | Low | Version 4 | Acknowledged

The functions `addDepositorsToAllowlist` and `removeDepositorsFromAllowlist` in `ERC20RootVault` restrict the access control with function `_requireAtLeastStrategy`. However, neither MStrategy nor LStrategy call these functions. Similarly, multiple functions in `VaultGovernance` use the same access control, although they are not called by the strategies.

Acknowledged:

Mellow Finance is aware that these functions are not called by smart contracts implementing the strategies, but they can be called by an EOA in case it manages the vault system. Client replied:

```
The vault system can be managed not by strategy, but by some account. In such a
case this account should have the possibility to edit `depositorsAllowList`.
These 2 functions exist for this reason.
```

5.7 Redundant Check for `baseSupply`

Design | Low | Version 4 | Acknowledged

The function `ERC20RootVault._chargePerformanceFees` performs a check of `baseSupply` is equal to 0, and returns if this is the case:

```
if ((performanceFee == 0) || (baseSupply == 0)) {
    return;
}
```

However, this check is redundant because `_chargeFees` performs the same check and returns before calling `_chargePerformanceFees`.

Acknowledged:

Client acknowledged the redundant check but has decided to keep it as it enhances the readability of the code.

5.8 Redundant Check for `deltaSupply`

Design | Low | Version 4 | Acknowledged

The function `_getBaseParamsForFees` performs the following check on withdrawals:

```
baseSupply = 0;
if (supply > deltaSupply) {
    baseSupply = supply - deltaSupply;
}
```

The `deltaSupply` corresponds to the LP shares that a user is burning, which is less than or equal to the balance of that user. Hence, it is always less or equal to the `totalSupply`.

Acknowledged:

Client acknowledged the redundant check but has decided to keep it as it enhances the readability of the code.

5.9 Redundant Checks on Push Function

Design | Low | Version 4 | Acknowledged

The function `IntegrationVault.push` performs the following checks that are always true when a vault is linked to a root vault:

```
uint256 nft_ = _nft;
require(nft_ != 0, ExceptionsLibrary.INIT);
IVaultRegistry vaultRegistry = _vaultGovernance.internalParams().registry;
IVault ownerVault = IVault(vaultRegistry.ownerOf(nft_)); // Also checks that the token exists
uint256 ownerNft = vaultRegistry.nftForVault(address(ownerVault));
require(ownerNft != 0, ExceptionsLibrary.NOT_FOUND);
```

Acknowledged:

Mellow Finance has decided to keep the checks to prevent from pushing and pulling on uninitialized vaults.

5.10 State Updates After Reentrancy Possibility

Design | Low | Version 4 | Code Partially Corrected

When creating a vault, `_mint` is called to mint the NFT. This calls the receiver and gives an opportunity to reenter the system.

```
_safeMint(owner, nft);
_vaultIndex[nft] = vault;
_nftIndex[vault] = nft;
_vaults.push(vault);
_topNft += 1;
emit VaultRegistered(tx.origin, msg.sender, nft, vault, owner);
```

State updates and events are emitted after the possible reentrancy in this function and the calling functions. Coding guidelines suggest following the check-effects-interaction pattern to mitigate reentrancy vulnerabilities.

Code partially corrected:

The minting statement `_safeMint` has been moved to the end of the function `registerVault`. However, state is still updated afterwards in functions `createVault` of vault governance contracts.

5.11 Missing Slippage Protection in `_mintNewNft`

Security Low Version 1 Acknowledged

The function `_mintNewNft` in `LStrategy` sets the parameters `amount0Min` and `amount1Min` of the `MintParams` to zero, hence disabling any slippage protection. However, the risk exposure in this case is limited as a new position in Uniswap should be open with small amounts `minTokenXForOpening`. The exact amount depends on `admin` who sets the `otherParams`.

Acknowledged:

Sanity checks were introduced in Version 3 to check if the variables `minTokenXForOpening` are smaller than `10**9`. This adds another layer of protection to ensure that the number of tokens is relatively low. Still, the number of tokens does not guarantee that the value is small.

5.12 `UniV3Vault` Pulls More Tokens Than Requested

Correctness Low Version 1 Acknowledged

`UniV3Vault._pullUniV3Nft` first calculates the amount of tokens to pull, then decreases the liquidity inside the Uniswap position and then collects the tokens. When the earnings have not been collected before, the last step additionally collects the earnings, returning more tokens than intended.

The function should take the tokens owed into consideration when calculating the amount to pull.

Acknowledged

Mellow Finance acknowledged the issue and replied that the strategy maintainer can call the `collectEarnings` function to collect all the fees.

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	5

- Mismatch of Specification With Uniswap V3 Oracle Code Corrected
- Chainlink Oracle Returns Empty Prices Code Corrected
- Incorrect LP Token Calculation in ERC20RootVault Code Corrected
- Missing Access Control in UniV3Oracle Code Corrected
- UniV3Oracle Returns Reverse Prices for Token Pairs Code Corrected

High-Severity Findings	14

- Incorrect TVL Conversion Code Corrected
- Adding up Total Value Locked on Different Tokens Code Corrected
- Calling _liquidityDelta Incorrectly Code Corrected
- Calling _liquidityDelta With Incorrect Inputs Code Corrected
- Incorrect Observation Index in _getAverageTick Code Corrected
- Incorrect Parameters on externalCall Code Corrected
- Insufficient Testing Code Corrected
- Opposite Vaults Are Swapped Code Corrected
- Possibility to Exit Positions of Any Address Code Corrected
- Possible DOS From First Depositor Code Corrected
- Setting Wrong State Variable Code Corrected
- Wrong Formula in _rebalanceUniV3Liquidity Code Corrected
- Wrong TVL Calculation in ERC20RootVault Code Corrected
- liquidity Gets Overwritten in the Loop Code Corrected

Medium-Severity Findings	17

- Wrong State Variable Updated Code Corrected
- Inconsistent Access Control for Rebalance in LStrategy Specification Changed
- Inconsistent Sanity Check on First Deposit's Amounts Code Corrected
- Safety Level of Returned Prices Can Silently Downgrade Specification Changed
- Unfair Distribution of LP Shares in ERC20RootVault Code Corrected
- Conflicting Specifications for MStrategy Specification Changed
- Implementation Differs From Specification on _targetTokenRatioD Code Corrected
- Incorrect Access of Addresses in EnumerableSet Code Corrected
- Missing Checks for Dust Amounts When Rebalancing Pools Code Corrected
- Missing Delay Restriction in BaseValidator Code Corrected

- Missing Sanity Checks in signOrder Code Corrected
- No Slippage Protection in Multiple Contracts Code Corrected
- Possible Underflow in UniV3Oracle.price Code Corrected
- Rebalance in LStrategy Can Leave Tokens in the Vault to Be Closed Code Corrected
- Subvault Tokens Are Not Checked in AggregateVault Code Corrected
- Transferring Tokens Only to lowerVault Code Corrected
- Use of Libraries Code Corrected

Low-Severity Findings	41

- Missing Sanity Checks for intervalWidthInTicks Code Corrected
- Possible Attack by First Depositor Specification Changed
- Possible Optimization on _chargePerformanceFees Code Corrected
- Possible Violation of the Minimum Token Amounts After the First Deposit Code Corrected
- Misleading Function Name and Natspec Code Corrected
- Mismatch of Specifications for StrategyParams Code Corrected
- Missing Sanity Check for maxSlippageD in MStrategy Code Corrected
- Missing Sanity Checks for oracleSafetyMask Code Corrected
- Possible Struct Optimization in Strategies Code Corrected
- Redundant Comparisons Code Corrected
- Redundant Storage Read in ERC20Vault._pull Code Corrected
- Variables Can Be Declared as Constant Code Corrected
- Incorrect Specification for reclaimTokens Specification Changed
- Missing Natspec Description for minDeviation Code Corrected
- Casting of maxTickDeviation Code Corrected
- Check Requirements First Code Corrected
- Duplicate Code _permissionIdsToMask Code Corrected
- Duplicate Storage Read in Deposit Code Corrected
- Inconsistent Specifications Specification Changed
- Inefficient Array Shrinking Code Corrected
- Inefficient State Variable Packing Code Corrected
- Misleading Naming of Variables in UniV3Oracle Code Corrected
- Missing Sanity Check in MStrategy.createStrategy Code Corrected
- Missing Sanity Checks for Params Code Corrected
- Misspelled Variable Names Code Corrected
- Possible Struct Optimization Code Corrected
- Rebalance in MStrategy Is Inconsistent Code Corrected
- Specification for minDeviation Not Enforced Code Corrected
- Storing Redundant Data in Storage Code Corrected
- Unnecessary Approval to Vault Registry Code Corrected

- Unused Constant in ERC20Validator Code Corrected
- Unused Event DeployedVault Code Corrected
- Unused Function LStrategy._priceX96FromTick Code Corrected
- Unused Imports Code Corrected
- Wrong Check of Minimum Token Amounts in ERC20RootVault.withdraw Code Corrected
- Wrong Specification for YearnVault.tvl Specification Changed
- ContractRegistry DOS Code Corrected
- ERC20Vault._pull Forces Push of Wrong Amount of Tokens Code Corrected
- IntegrationVault._root Does Not Check the NFT of the Root Vault Code Corrected
- VaultGovernance.commitInternalParams Does Not Delete Staged Parameters Code Corrected
- registry.ownerOf Is Called Twice in IntegrationVault.pull Code Corrected

6.1 Mismatch of Specification With Uniswap V3 Oracle

Correctness Critical Version 2 Code Corrected

The specifications of the function `price` for oracles are in the interface `IOracle` as following:

```
/// @dev The price is token1 / token0 i.e. how many weis of token1
/// required for 1 wei of token0.

function price(
    address token0,
    address token1,
    uint256 safetyIndicesSet
) external view returns (uint256[] memory, uint256[] memory);
```

According to the specification, `priceA_B = price(tokenA, tokenB)` should be the inverse of `priceB_A = price(tokenB, tokenA)`, meaning the following relation should hold: `priceA_B = 1 / priceB_A`.

The function `UniV3Oracle.price` in Version 2 returns the same price for a pair of tokens without differentiating in which denomination token the price should be. Namely, the function returns the same prices when calling `price(tokenA, tokenB)` or `price(tokenB, tokenA)`. This behavior is enforced in the first `if` statement of the function:

```
if (token0 > token1) {
    (token0, token1) = (token1, token0);
}
```

Code corrected:

The Uniswap V3 Oracle has been revised, the Uniswap's OracleLibrary is now used and a flag `isSwapped` is added to track the correct denomination of the returned price.

6.2 Chainlink Oracle Returns Empty Prices

Correctness Critical Version 1 Code Corrected

`ChainlinkOracle` maintains the mapping `oraclesIndex` which stores addresses of chainlink oracles for each token. The mapping is populated by the `admin` through the function `_addChainlinkOracles`:

```
function _addChainlinkOracles(address[] memory tokens, address[] memory oracles) internal {
    ...
    oraclesIndex[token] = oracle;
    ...
}
```

The function `price(token0,token1,safetyIndicesSet)` checks if the mapping `oraclesIndex` has the addresses for the respective Chainlink oracles:

```
if ((address(chainlinkOracle0) != address(0)) || (address(chainlinkOracle1) != address(0))) {
        return (pricesX96, safetyIndices); // returns empty values
}
```

The condition above is incorrect as it returns empty values if the Chainlink oracles exist in the mapping. This makes the Chainlink oracle - assumed to be the safest by the specifications and the code - unusable.

Code corrected:

The above check in function `price` has been revised to return empty prices only if there is no entry for at least one of the tokens in mapping `oraclesIndex`:

```
 if ((address(chainlinkOracle0) == address(0)) || (address(chainlinkOracle1) == address(0))) {
    return (pricesX96, safetyIndices);
}
```

6.3 Incorrect LP Token Calculation in `ERC20RootVault`

Correctness Critical Version 1 Code Corrected

`ERC20RootVault._getLpAmount` incorrectly calculates the minimum of given token amounts. An attacker can issue more LP tokens than he is entitled to and can then exchange them back for additional tokens.

The following code incorrectly resets the MIN calculation for as many iterations as `tokenLpAmount` is equal to 0:

```
for (uint256 i = 0; i < tvlsLength; ++i) {
    if ((amounts[i] == 0) || (tvl_[i] == 0)) {
        continue;
    }

    uint256 tokenLpAmount = FullMath.mulDiv(amounts[i], supply, tvl_[i]);
    if ((tokenLpAmount < lpAmount) || (lpAmount == 0)) {
        lpAmount = tokenLpAmount;
```

```
    }
}
```

If `tokenLpAmount == 0` in the first iteration, `lpAmount` will be set to 0. If `tokenLpAmount > 0` in the next iteration, `lpAmount` will be set to `tokenLpAmount` although it is larger than the already set value.

In a later step, `ERC20RootVault._getNormalizedAmount` normalizes the sent token amounts to the calculated `lpAmount`. This function however does not increase the normalized amount to a value greater than the sent one. An attacker can therefore exploit this by calling `deposit` with all token amounts but the last one being set to 0 and then calling `withdraw` with the LP tokens that have just been minted to obtain his initial investment plus an amount of all other tokens in the `Vault` equal to the current ratio of tokens.

Code corrected:

The function `_getLpAmount` has been refactored to set the `lpAmount` to the minimum of `tokenLpAmount` calculated on each iteration of the `for loop`. The flag `isLpAmountUpdated` is set to `true` on the first iteration that a non-zero value is assigned to `lpAmount`.

6.4 Missing Access Control in `UniV3Oracle`

Security | Critical | Version 1 | Code Corrected

The function `addUniV3Pools` populates the mapping `poolsIndex` with the address of a Uniswap pool for a pair of tokens. The function should be accessible only to trusted accounts, however, it does not implement any access restriction. As the function is `external` anyone can set arbitrary addresses as Uniswap pools, hence freely manipulate the oracle prices.

Code corrected:

The updated code resolves the issue by restricting the access to the function `addUniV3Pools` only to the admin, hence preventing malicious users from setting arbitrary addresses as Uniswap pools:

```
function addUniV3Pools(IUniswapV3Pool[] memory pools) external {
    _requireAdmin();
    _addUniV3Pools(pools);
}
```

6.5 `UniV3Oracle` Returns Reverse Prices for Token Pairs

Correctness | Critical | Version 1 | Code Corrected

The `UniV3Oracle` computes the price for two tokens using the Uniswap V3 observations. As the tokens in Uniswap are always sorted by their address (Token0 < Token1), the function `price` uses a flag `revTokens` to distinguish if the price from Uniswap corresponds to the order of function parameters, or if it should be reversed. The respective code is:

```
function price(address token0,address token1,uint256 safetyIndicesSet)
    external view returns (uint256[] memory pricesX96, uint256[] memory safetyIndices) {
   ...
    bool revTokens = token1 > token0;

    for (uint256 i = 0; i < len; i++) {
        if (revTokens) {
            pricesX96[i] = FullMath.mulDiv(CommonLibrary.Q96, CommonLibrary.Q96, pricesX96[i]);
        }
        pricesX96[i] = FullMath.mulDiv(pricesX96[i], pricesX96[i], CommonLibrary.Q96);
    }
}
```

The flag `revToken` is set to `true` if the tokens in the function parameters are ordered as in Uniswap, hence incorrectly reverses the computed price.

Code corrected:

The contract `UniV3Oracle` has been refactored due to the bug presented above and other issues reported for this contract. The code above that mistakenly reversed the prices is not present anymore in Version 2, however, another issue has been introduced on the fix.

6.6 Incorrect TVL Conversion

Correctness | High | Version 3 | Code Corrected

The function `_getTvlToken0` incorrectly converts the TVL amount of a given token `i` into token `0`. The oracle returns a price in `x96` format. This price is directly used as if it would be a correctly formatted price to convert the amounts. As the TVL in most cases will be lower than the price in `x96` format the calculation will return 0.

```
tvl0 = tvls[0];
for (uint256 i = 1; i < tvls.length; i++) {
    (uint256[] memory prices, ) = oracle.price(tokens[0], tokens[i], 0x28);
    require(prices.length > 0, ExceptionsLibrary.VALUE_ZERO);
    uint256 price = 0;
    for (uint256 j = 0; j < prices.length; j++) {
        price += prices[j];
    }
    price /= prices.length;
    tvl0 += tvls[i] / price;
```

Additionally, the calculation would be more precise if the price would be multiplied to convert the amounts.

Code corrected:

The issue about the conversion of TVLs in function `_getTvlToken0` has been addressed. The last statement of the `for-loop` has been changed:

```
tvl0 += FullMath.mulDiv(tvls[i], CommonLibrary.Q96, priceX96);
```

6.7 Adding up Total Value Locked on Different Tokens

Correctness High Version 1 Code Corrected

The function `postPreOrder` calls the function `_liquidityDelta` with `tvl[0]` and `tvl[0] + tvl[1]` (see the issue reported in Calling _liquidityDelta incorrectly).

Additionally, the calculations are performed on `tvl` with different underlying tokens. Namely, `tvl[0]` is in the denomination of `token0`, while `tvl[1]` in the denomination of `token1`.

Code corrected:

The issue is resolved in code base Version 2, the first argument `tvl[0]` is converted into the domination of `token1` before passed to `_liquidityDelta`, while the second parameter `tvl[1]` remains in the denomination of `token1`.

6.8 Calling `_liquidityDelta` Incorrectly

Correctness High Version 1 Code Corrected

The function `postPreOrder` in `Lstrategy` calls `_liquidityDelta` as follows:

```
(uint256 tokenDelta, bool isNegative) = _liquidityDelta(
    tvl[0],
    tvl[0] + tvl[1],
    ratioParams.erc20TokenRatioD,
    ratioParams.minErc20TokenRatioDeviationD
);
```

As already pointed out in the issue Calling _liquidityDelta with incorrect inputs, the function `_liquidityDelta` also performs the addition, hence computing incorrectly the result.

Code corrected:

The parameters passed to the function `_liquidityDelta` have been corrected, namely the addition of `tvl[0] + tvl[1]` is removed and only `tvl[1]` is passed as the second argument of the function call.

6.9 Calling `_liquidityDelta` With Incorrect Inputs

Correctness High Version 1 Code Corrected

The function `rebalanceERC20UniV3Vaults` in `LStrategy` calls `_liquidityDelta` as follows:

```
(capitalDelta, isNegativeCapitalDelta) = _liquidityDelta(
    erc20VaultCapital,
    erc20VaultCapital + lowerVaultCapital + upperVaultCapital,
    ratioParams.erc20UniV3CapitalRatioD,
```

```
    ratioParams.minErc20UniV3CapitalRatioDeviationD
);
```

Note that, the first parameter is included in the sum used as the second parameter. However, the function `_liquidityDelta` also performs the addition on the code below, hence computing `targetLowerLiquidity` incorrectly:

```
uint256 targetLowerLiquidity = FullMath.mulDiv(
    targetLiquidityRatioD,
    lowerLiquidity + upperLiquidity,
    DENOMINATOR
);
```

Code corrected:

In `rebalanceERC20UniV3Vaults` the calculation does not add `erc20VaultCapital` anymore.

6.10 Incorrect Observation Index in `_getAverageTick`

Correctness | High | Version 1 | Code Corrected

Function `_getAverageTick` computes the `averageTick` and the `tickDeviation` based on the most recent observation and a previous observation referred as `observationIndexLast`. The latter index is computed as follows:

```
uint16 observationIndexLast = observationIndex >= oracleObservationDelta
    ? observationIndex - oracleObservationDelta
    : observationIndex + (type(uint16).max - oracleObservationDelta + 1);
```

If `oracleObservationDelta` is larger than `observationIndex` (e.g., by 1), the code above returns a value that is close (or equal) to `type(uint16).max`. It is very likely that the Uniswap pool has a smaller cardinality of observations than the computed `observationIndexLast`, hence 0s would be returned for this observation.

Code corrected:

The formula to compute `observationIndexLast` when `oracleObservationDelta > observationIndex` has been revised, `type(uint16).max` has been replaced with `observationCardinality`.

obsIdx = 20 delta = 30 card = 50 --- 20 + 50 -30 = 40

obsIdx = 30 delta = 30 card = 50 --- 0

obsIdx = 30 delta = 31 card = 50 --- 30 + 50 -31 = 49

obsIdx = 30 delta = 49 card = 50 --- 30 + 50 - 49 = 31

generalized: obsIdx + card - delta % card

6.11 Incorrect Parameters on `externalCall`

Correctness High Version 1 Code Corrected

The function `signOrder` in `LStrategy` performs few `externalCall`s, and for one of them sets the wrong parameters as input:

```
bytes memory setPresignatureData = abi.encode(SET_PRESIGNATURE_SELECTOR, uuid, signed);
erc20Vault.externalCall(cowswap, SET_PRESIGNATURE_SELECTOR, setPresignatureData);
```

Note that the function selector is part of the `abi.encode` and then is set as the second parameter in `externalCall`, which also appends the selector when executing the call, hence causing the external function to always fail:

```
(bool res, bytes memory returndata) = to.call{value: msg.value}(abi.encodePacked(selector, data));
```

Code corrected:

The external call in `LStrategy.signOrder` does not encode the `SET_PRESIGNATURE_SELECTOR` twice anymore.

6.12 Insufficient Testing

Security High Version 1 Code Corrected

We found an unusual high number of issues that would have been easily detected with proper tests. The current unit and integration tests are insufficient.

Code corrected:

The tests have been extended significantly on the latest iterations of the review process to cover more functions and call paths.

6.13 Opposite Vaults Are Swapped

Correctness High Version 1 Code Corrected

The function `_swapVaults` in `LStrategy` should close the position with no liquidity and open a new one given the price move in `positiveTickGrowth`. The decision on which vault to close is done in the following `if` condition:

```
/// @param positiveTickGrowth `true` if price tick increased
...
if (!positiveTickGrowth) {
    (fromVault, toVault) = (lowerVault, upperVault);
} else {
    (fromVault, toVault) = (upperVault, lowerVault);
}
```

The function closes the `fromVault` and creates the new vault according to the current position of `toVault`. However, the code above assigns `fromVault` wrongly to `lowerVault` if the tick is

decreasing, and vice-versa if the tick is increasing. Given this error and the following requirement, the function would fail always (as `fromVault` has all liquidity):

```
require(fromLiquidity == 0, ExceptionsLibrary.INVARIANT);
```

Code corrected:

The vaults were switched like:

```
if (!positiveTickGrowth) {
        (fromVault, toVault) = (upperVault, lowerVault);
    } else {
        (fromVault, toVault) = (lowerVault, upperVault);
    }
```

6.14 Possibility to Exit Positions of Any Address

Security High Version 1 Code Corrected

In `ERC20RootVault.withdraw`, LP tokens are burned in a call to `_burn` from the address that is specified in the `to` parameter. Neither `_burn` nor any other statement in `withdraw` performs access control checks to verify if the `msg.sender` is allowed to burn the tokens of the given address. Thus, any user can burn LP tokens of a given address and transfer the underlying tokens to that address.

Finally, an incorrect event is emitted with `msg.sender`.

Code corrected:

The issues have been resolved in the updated code Version 2. The function `withdraw` now burns only the LP tokens of the `msg.sender`, while transfers the underlying tokens to the address `to` specified by the caller.

6.15 Possible DOS From First Depositor

Security High Version 1 Code Corrected

The first user that calls `deposit` in `ERC20RootVault` can choose freely any amount (including zero) for each vault token, while the LP shares are set to the largest amount by the following loop in `_getLpAmount`:

```
for (uint256 i = 0; i < tvl_.length; ++i) {
    if (amounts[i] > lpAmount) {
        lpAmount = amounts[i];
    }
}
```

However, if the first user (on initialization or whenever `totalSupply` is zero) chooses to deposit only one token (e.g., `token[0]`) it makes impossible for other users to deposit other tokens (e.g., `token[1]`) as the `totalSupply` is not zero anymore, and `_getNormalizedAmount` considers the existing TVL:

```
// normalize amount
uint256 res = FullMath.mulDiv(tvl_, lpAmount, supply); // if tvl_ == 0, res = 0
```

The intended use of the function might be that the first deposit is done by a trusted account, but this is not enforced.

Code corrected:

A new constant `FIRST_DEPOSIT_LIMIT` is introduced and a require checks that each token amount is above this limit with `tokenAmounts[i] > FIRST_DEPOSIT_LIMIT`.

6.16 Setting Wrong State Variable

Correctness **High** **Version 1** **Code Corrected**

The function `_setOperatorParams` in `VaultGovernance`, as the name suggests, should update the state variable `_operatorParams`, instead it overwrites the variable `_protocolParams`:

```
function _setOperatorParams(bytes memory params) internal {
    _requireAtLeastOperator();
    _protocolParams = params;
}
```

This mistake has severe consequences: operator gets admin privileges to set `_protocolParams` or can set a vault state to incorrect parameters. Finally, the functionality to initialize or update the `_operatorParams` is missing.

Code corrected:

The issue is resolved and now the function `_setOperatorParams` sets the operator params as intended. The natspec description has been updated accordingly also.

6.17 Wrong Formula in `_rebalanceUniV3Liquidity`

Correctness **High** **Version 1** **Code Corrected**

The function `_rebalanceUniV3Liquidity` in `LStrategy` updates the value of `liquidity` as follows:

```
liquidity = uint128(
    FullMath.mulDiv(
        availableBalances[i],
        shouldDepositTokenAmountsD[i] - shouldWithdrawTokenAmountsD[i],
        DENOMINATOR
    )
);
```

The formula above is wrong, it multiplies two amounts in `token[i]`, then divides the result with `DENOMINATOR`.

Code corrected:

The formula now multiplies with `DENOMINATOR` and divides by the token amount.

6.18 Wrong TVL Calculation in `ERC20RootVault`

Correctness | High | Version 1 | Code Corrected

`ERC20RootVault._getTvlToken0` calculates the TVL of the `Vault` denominated in the token at position 0 of an array of tokens. It iterates over all the tokens in the array, but only ever compares token with index 0 to token with index 1. It should, however, compare token with index 0 to the token with the current iteration's index. The function is only used in `_calculatePerformanceFees`.

```
for (uint256 i = 1; i < tvls.length; i++) {
        (uint256[] memory prices, ) = oracle.price(tokens[0], tokens[1], 0x28);
```

Code corrected:

The issue has been resolved as the correct index is now used when querying the price of tokens inside the loop.

6.19 `liquidity` Gets Overwritten in the Loop

Correctness | High | Version 1 | Code Corrected

The following loop in `LStrategy._rebalanceUniV3Liquidity` updates the liquidity for vault tokens in a loop:

```
for (uint256 i = 0; i < 2; i++) {
    ...
    liquidity = uint128(
        FullMath.mulDiv(
            availableBalances[i],
            shouldDepositTokenAmountsD[i] - shouldWithdrawTokenAmountsD[i],
            DENOMINATOR
        )
    );
}
```

The final value of `liquidity` after the loop exists should be the minimum value calculated in each iteration, however, the loop above overwrites the `liquidity` on each iteration without performing any check.

Code corrected:

In Version 2 the `potentialLiquidity` is computed on each iteration of the loop and it is compared with `liquidity`, hence `liquidity` can only decrease in the loop:

```
liquidity = potentialLiquidity < liquidity ? potentialLiquidity : liquidity;
```

6.20 Wrong State Variable Updated

Design Medium Version 8 Code Corrected

The function `LStrategy.rebalanceUniV3Vaults` updates the wrong state variable when storing the timestamp of the ongoing rebalance:

```
require(
    block.timestamp >= lastRebalanceUniV3VaultsTimestamp + otherParams.secondsBetweenRebalances,
    ExceptionsLibrary.TIMESTAMP
);
lastRebalanceERC20UniV3VaultsTimestamp = block.timestamp;
```

Due to this error the throttling mechanism does not work as expected for the function rebalancing the two uniswap vaults. Furthermore, this also affects the throttling mechanism of the function `rebalanceERC20UniV3Vaults`.

Code corrected:

The issue has been fixed and the correct state variable is updated in `rebalanceUniV3Vaults`:

```
lastRebalanceUniV3VaultsTimestamp = block.timestamp;
```

6.21 Inconsistent Access Control for Rebalance in LStrategy

Design Medium Version 4 Specification Changed

The function `LStrategy.rebalanceERC20UniV3Vaults` restricts the access to only accounts with operator or admin roles. However, functions `deposit` and `withdraw` in the `ERC20RootVault` do not have any access restriction (unless the vault is private). The root vault has the `operator` role in LStrategy and for any deposit or withdraw operation, the vault triggers the rebalance function in LStrategy, hence circumventing the access control of the rebalance function.

Specification changed:

Mellow Finance has decided to remove the callback feature that triggered the rebalance in `LStrategy`. Now, the rebalance functions `rebalanceERC20UniV3Vaults` and `rebalanceUniV3Vaults` can be called only by whitelisted addresses with either `admin` or `operator` role. Note that, the callback feature is still present in `ERC20RootVault` in case future strategies will support the callback feature.

6.22 Inconsistent Sanity Check on First Deposit's Amounts

Design Medium Version 4 Code Corrected

The function `ERC20RootVault.deposit` runs the following loop for the first deposit (whenever `totalSupply` is `0`) to check that all amounts are above a threshold `FIRST_DEPOSIT_LIMIT` (hard-coded to `10000`):

```
if (totalSupply == 0) {
    for (uint256 i = 0; i < tokens.length; ++i) {
        require(tokenAmounts[i] > FIRST_DEPOSIT_LIMIT, ExceptionsLibrary.LIMIT_UNDERFLOW);
    }
}
```

The contract uses another set of thresholds per token `_pullExistentials` which are initialized as: `10**(token.decimals() / 2)`. Hence for tokens with more than 8 decimals, there is a gap between the two thresholds `FIRST_DEPOSIT_LIMIT` and `_pullExistentials`. If the first deposit includes an amount for a token in this gap, the contract does not allow new deposits for the token from other users as the respective TVL will be always below the threshold `_pullExistentials`. This behavior is enforced in `_getLpAmount`:

```
for (uint256 i = 0; i < tvlsLength; ++i) {
    if (tvl_[i] < pullExistentials[i]) {
        continue;
    }
    ...
}
```

and in the function `_getNormalizedAmount`:

```
if (tvl_ < existentialsAmount) {
    // use zero-normalization when all tvls are dust-like
    return 0;
}
```

Code corrected:

Mellow Finance now requires that the amount in the first deposit is 10 times the `_pullExistentials`.

6.23 Safety Level of Returned Prices Can Silently Downgrade

Security Medium Version 4 Specification Changed

The function `UniV3Oracle.price` returns more than one price depending on the value of `safetyIndicesSet`. UniV3Oracle supports 4 safety levels:

- Safety level 1: spot price.
- Safety level 2: average price based on observations from last `2.5` minutes.

- Safety level 3: average price based on observations from last `7.5` minutes.
- Safety level 4: average price based on observations from last `30` minutes.

If a Uniswap pool does not have enough observations required for a safety level, the oracle skips the prices for such safety levels and returns only prices with lower safety levels. The respective code:

```
for (uint256 i = 2; i < 5; i++) {
    ...
    (int24 tickAverage, , bool withFail) = OracleLibrary.consult(address(pool), observationTimeDelta);
    if (withFail) {
        break;
    }
    ...
}
```

Specifications changed:

The natspec description of `IOracle.priceX96` has been updated to be more explicit about this behavior:

```
/// @notice It is possible that not all indices will have their respective prices returned.
```

Also, more detailed description has been added in `UniV3Oracle.priceX96`:

```
/// If there is no initialized pool for the passed tokens, empty arrays will be
    returned.
/// Depending on safetyIndicesSet if the 1st bit in safetyIndicesSet is non-zero, then
    the response will contain the spot price.
/// If there is a non-zero 2nd bit in the safetyIndicesSet and the corresponding
    position in the pool was created no later than |l|_OBS_DELTA seconds ago,
/// then the average price for the last |l|_OBS_DELTA seconds will be returned. The
    same logic exists for the 3rd and MID_OBS_DELTA, and 4th index and |hl|_OBS_DELTA.
```

6.24 Unfair Distribution of LP Shares in ERC20RootVault

Design | Medium | Version 3 | Code Corrected

The `ERC20RootVault` charges the management, protocol and performance fees by minting new LP shares, hence inflating the total supply. The function `_chargeFees` is triggered on every deposit (and withdraw) action, hence the total supply of LP shares after a deposit increases more than the amount of LP shares awarded to the depositor. In this way, a second deposit of the same token amounts after the fees have been charged, receives more LP shares than the first one.

For example, assume that the `ERC20RootVault` has been initialized and a first user deposits `10` `TokenA` and `10` `TokenB` (assuming 0 decimals for simplicity) and receives `10` LP shares. As the fees will be charged on deposit, let's suppose another `1` LP share will be minted, hence in total there are `11` LP shares minted after the deposit. If a second user deposits the same amounts `10` `TokenA` and `10` `TokenB`, the function `_getLpAmount` will award `11` LP shares to the user although the same amounts were deposited.

Code corrected:

The issue has been addressed by modifying the functions `deposit` to charge fees first and then compute the LP shares awarded to the user according to the new LP supply.

6.25 Conflicting Specifications for MStrategy

Correctness Medium Version 1 Specification Changed

The specifications of MStrategy have conflicting instructions. The section "TickMin and TickMax update" states:

```
tickMin and tickMax are initially set to some ad-hoc params.
As soon as the current price — tick is greater than tickMax - tickNeiborhood
or less than tickMin + tickNeiborhood  the boundaries of the interval
is expanded by tickIncrease amount.
```

In the rebalance steps, `tickNeiborhood` is used instead of `tickIncrease`:

```
- tick is greater than tickMax - tickNeiborhood then new
    boundaries are [tickMin, tickMax + tickNeiborhood]

- tick is less than tickMin + tickNeiborhood then new
    boundaries are [tickMin - tickNeiborhood, tickMax]
```

Specification changed:

The specification was changed accordingly.

6.26 Implementation Differs From Specification on `_targetTokenRatioD`

Correctness Medium Version 1 Code Corrected

The specifications use the following formula to compute the portions of tokens in a Uniswap v3 pool: |

$$w_x = \frac{tick - tickMax}{tickMin - tickMax}$$

However, the implementation uses the following code:

```
return (uint256(uint24(tick - tickMin)) * DENOMINATOR) / uint256(uint24(tickMax - tickMin));
```

which corresponds to the following formula: |

$$w_x = \frac{tick - tickMin}{tickMax - tickMin}$$

Code corrected:

The implementation of `MStrategy._targetTokenRatioD` has been updated to comply to the specification.

6.27 Incorrect Access of Addresses in `EnumerableSet`

Correctness Medium Version 1 Code Corrected

Function `commitAllValidatorsSurpassedDelay` in the protocol governance contract has a `for` loop that iterates through `_stagedValidatorsAddresses` and commits the ones for which the delay period has passed. The respective code is:

```
for (uint256 i; i != length; i++) {
    address stagedAddress = _stagedValidatorsAddresses.at(0);
    if (block.timestamp >= stagedValidatorsTimestamps[stagedAddress]) {
        ...
    }
}
```

The variable `stagedAddress` inside the loop points always to the hard-coded index `0`, hence if there is at least one address in staged validators for which the deadline has not passed, the loop will just run until it reaches `i==length`.

Code corrected:

The `0` was replaced by the index variable `i`. The loop exit conditions were changed to:

```
uint256 length = _stagedValidatorsAddresses.length();
...
uint256 addressesCommittedLength;
for (uint256 i; i != length;) {
        address stagedAddress = _stagedValidatorsAddresses.at(i);
            ...
            addressesCommitted[addressesCommittedLength] = stagedAddress;
            ++addressesCommittedLength;
            --length;
            ...
        } else {
            ++i;
        }
```

6.28 Missing Checks for Dust Amounts When Rebalancing Pools

Design Medium Version 1 Code Corrected

The function `_rebalancePools` in `MStrategy` rebalances the `erc20Vault` and `moneyVault` to comply to the specified ratio `erc20MoneyRatioD`. The rebalancing is performed always when a non-zero amount should be moved from one vault to the other, i.e., even for dust amounts. Considering that `pull` is relatively costly, the strategy would be more efficient if it performs the rebalancing of the two pools only if a minimum threshold of tokens should be moved.

Code corrected:

The updated code does not perform the token transfers if only dust amounts should be moved:

```
if ((absoluteTokenAmounts[0] < minDeviation) && (absoluteTokenAmounts[1] < minDeviation)) {
    return tokenAmounts;
}
```

6.29 Missing Delay Restriction in `BaseValidator`

Correctness Medium Version 1 Code Corrected

Setting the new params in `BaseValidator` follows the pattern stage-wait-commit. On staging the new parameters, the respective timestamp is updated:

```
_stagedValidatorParamsTimestamp = block.timestamp + governance.governanceDelay;
```

However, the `admin` of the `governance` can commit the staged parameters at any time, e.g., immediately after staging them, by calling `commitValidatorParams` as the function does not check if the delay period has passed.

Code corrected:

The function now checks the delay with a require validating `block.timestamp >= _stagedValidatorParamsTimestamp`.

6.30 Missing Sanity Checks in `signOrder`

Security Medium Version 1 Code Corrected

The function `signOrder` in `LStrategy` performs some sanity checks if the submitted `order` is in line with the values of the posted `preOrder`. However, the check for `order.receiver` is missing, therefore the caller can set any arbitrary address and receive the `buyToken`.

Code corrected:

The code doing the sanity checks for `order` in `signOrder` has been moved to the separate function `LStrategyOrderHelper.checkOrder` which includes the check that the `receiver` is the `erc20Vault`.

6.31 No Slippage Protection in Multiple Contracts

Security Medium Version 1 Code Corrected

`push` and `pull` functions in `UniV3Vault` take options arguments that contain the minimum amount of tokens for slippage protection.

`push` and `pull` functions in `MellowVault` take an options argument that contains the minimum amount of LP tokens for slippage protection.

In the following cases, these options are not used:

- `ERC20RootVault.deposit` calls `AggregateVault._push` without options, which could result in a call to `_push` of one of the described `Vault``s without slippage protection if the first ``subVault of the `ERC20RootVault` is one of the described `Vault` s. With the current contract setup, this is not possible though.
- `ERC20RootVault.withdraw` calls `AggregateVault._pull` without options, which could result in a call to `_pull` of one of the described ``Vault``s without slippage protection.
- `MStrategy.manualPull` calls `pull` of an arbitrary `Vault` without options, which could result in a call to `_pull` of one of the described ``Vault``s without slippage protection.
- `MStrategy._rebalancePools` calls `pull` of an arbitrary `Vault` without options, which could result in a call to `_pull` of one of the described ``Vault``s without slippage protection.
- `MStrategy._swapToTarget` calls `pull` of an arbitrary `Vault` without options, which could result in a call to `_pull` of one of the described ``Vault``s without slippage protection.

Code corrected:

A new parameter with option for slippage protection was introduced.

6.32 Possible Underflow in `UniV3Oracle.price`

Correctness | Medium | Version 1 | Code Corrected

The `UniV3Oracle` computes the price of two tokens based on two observations `obs1` and `obs0` from the Uniswap. The respective code is:

```
uint256 obs1 = (uint256(observationIndex) + uint256(observationCardinality) - 1) %
               uint256(observationCardinality);
uint256 obs0 = (uint256(observationIndex) + uint256(observationCardinality) - bfAvg) %
               uint256(observationCardinality);
int256 tickAverage;
{
    (uint32 timestamp0, int56 tick0, , ) = IUniswapV3Pool(pool).observations(obs0);
    (uint32 timestamp1, int56 tick1, , ) = IUniswapV3Pool(pool).observations(obs1);
    uint256 timespan = timestamp1 - timestamp0; // reverts
    ...
}
```

The `obj1` points to the previous observation (the one before the most recent observation), while the `obj0` should point to `bfAvg` observations before `obj1`. However, in case:

```
bfAvg == observationCardinality
```

`obj0` would point to the most recent observation, which would have a more recent timestamp than `obj1`, hence the statement to compute `timespan` would cause an underflow which reverts.

Code corrected:

The possibility of the underflow as described above has been mitigated in the updated code as the `bfAvg` cannot be equal to `obersvationCardinality`:

```
if (observationCardinality <= bfAvg) {
    continue;
}
```

Note that, the oracle does not return a price if for some pool `bfAvg` is equal to the observations cardinality.

6.33 Rebalance in LStrategy Can Leave Tokens in the Vault to Be Closed

Design Medium Version 1 Code Corrected

The internal function `_rebalanceUniV3Liquidity` should move the `desiredLiquidity` from one vault to the other depending on the price trend. If the price moves outside the range covered by a vault, all liquidity should be moved to the other vault and a new position should be open. However, given that `lowerVault` and `upperVault` operate on different price ranges, it means that they have different token ratios. Hence, when moving tokens from one vault to the other, the function caps the liquidity being transferred to the available balance in the cash position that can fill the token difference of two positions (the relevant code is shown below). However, if the cash position has insufficient balance to cover the difference for the whole liquidity being transferred, `fromVault` will have some remaining liquidity, hence it cannot be closed. As a consequence, a new Uniswap position cannot be created to cover the price as intended.

```
uint128 potentialLiquidity = uint128(
    FullMath.mulDiv(
        availableBalances[i],
        DENOMINATOR,
        shouldDepositTokenAmountsD[i] - shouldWithdrawTokenAmountsD[i]
    )
);
liquidity = potentialLiquidity < liquidity ? potentialLiquidity : liquidity;
```

Code corrected:

The function `LStrategy._rebalanceUniV3Liquidity` has been modified in Version 3 to withdraw everything from a vault when `desiredLiquidity` is set to maximum value of `uint128`, which is the case when a vault is to be closed. The relevant code is:

```
uint256[] memory withdrawTokenAmounts = fromVault.liquidityToTokenAmounts(
    desiredLiquidity == type(uint128).max ? desiredLiquidity : liquidity
);
pulledAmounts = fromVault.pull(
    address(erc20Vault),
    tokens,
    withdrawTokenAmounts,
    _makeUniswapVaultOptions(minWithdrawTokens, deadline)
);
```

The array `withdrawTokenAmounts` will have huge amounts when the `desiredLiquidity` is set to max `uint128`, but the `pull` operation is capped to the existing balance of the `fromVault`.

6.34 Subvault Tokens Are Not Checked in `AggregateVault`

Correctness **Medium** **Version 1** **Code Corrected**

`AggregateVault` requires the `_vaultTokens` state array to be initialized with the same tokens and the same ordering all the subvaults have been initialized with. However, this is not enforced upon initialization.

Code corrected:

When initializing, the vault of the nft is queried in `AggregateVault.initialize`. The vault's tokens are queried afterwards with the call `IIntegrationVault(vault).vaultTokens()`. A loop checks for each token in the vault if it matches the tokens from the initialization arguments.

6.35 Transferring Tokens Only to `lowerVault`

Correctness **Medium** **Version 1** **Code Corrected**

The following code should transfer tokens from `erc20Vault` to the two Uniswap vaults with the respective amounts:

```
if (!isNegativeCapitalDelta) {
    totalPulledAmounts = erc20Vault.pull(
        address(lowerVault),
        tokens,
        lowerTokenAmounts,
        _makeUniswapVaultOptions(minLowerVaultTokens, deadline)
    );
    pulledAmounts = erc20Vault.pull(
        address(lowerVault),
        tokens,
        upperTokenAmounts,
        _makeUniswapVaultOptions(minUpperVaultTokens, deadline)
    );
    for (uint256 i = 0; i < 2; i++) {
        totalPulledAmounts[i] += pulledAmounts[i];
    }
}
```

Both transfers above are from the `erc20Vault` to the `lowerVault`, hence no tokens are transferred to the `upperVault`.

Code corrected:

The bug has been fixed, the code now transfers the respective amounts to the `lowerVault` and `upperVault`.

6.36 Use of Libraries

Design | Medium | Version 1 | Code Corrected

Mellow Finance often uses own custom code for which battle proof libraries exist. We highly recommend using libraries instead of custom implementations. Especially, when dealing with complex DeFi projects like Uniswap V3.

Code Corrected:

The code part were most issues were found was the Uniswap oracle. In Version 3 Mellow Finance switched to the libraries provided by uniswap to interact with the oracle.

6.37 Missing Sanity Checks for `intervalWidthInTicks`

Design | Low | Version 5 | Code Corrected

The function `LStrategy.updateOtherParams` does not perform any sanity check on the `intervalWidthInTicks`. However, this parameter should be carefully updated as it affects directly the tick ranges covered by the two Uniswap vaults. For example, if the new width in ticks is the half of the existing one, the range of the new position would be fully covered by the existing vault (created with old width).

Code corrected:

In the updated version of the codebase, the parameter `intervalWidthInTicks` is declared as an `immutable` state variable, hence it set in the constructor and cannot be updated later.

6.38 Possible Attack by First Depositor

Security | Low | Version 5 | Specification Changed

The decimals of the LP shares distributed by root vaults are implicitly determined by the token amounts deposited by the first user. If the `totalSupply` ever goes to zero, or all TVLs are not significant, the next user that performs a deposit would affect the decimals of LP shares. This setup allows the first depositor to front-run and potentially exploit the next user depositing into the root vault. Consider the following example.

1. First Depositors deposits 10 WBTC (8 decimals, so 10**9 wei) and 10**-9 DAI (18 decimals, so 10**9 wei)
    - Receives 10**9 LP Tokens (= max(10**9, 10**9))
2. Second Depositor also sends a transaction to deposit 10 WBTC and 10**-9 DAI
    - Expects to receive also 10**9 LP Tokens, hence sets `minLpTokens` = 10**9
3. First depositor front-runs the transaction and performs these actions:

- withdraw() => withdraws everything, no fees charged
- deposit() => deposit 10**5 WBTC wei and 10**10 DAI wei => Receives 10**10 LP tokens
- withdraw() => withdraws ~ 9 * 10**9 LP => TVLs = [10**4 - 1 WBTC wei, 10**9 - 1 USDC wei]
- First depositor still has ~ 10**9 LP

4. Transaction of second depositor is executed
    - _getLpAmount -> isSignificantTvl == False
    - Receives 10**9 LP tokens => slippage protection passes
    - Deposits 10 WBTC and 10**-9 DAI
5. First depositor withdraws their ~ 10**9 LP and receives ~ 5 WBTC (after depositing only 0.0001 WBTC)

Specifications changed:

The updated code mitigates the attack presented above by enforcing the first deposit into a root vault to mint LP shares to `address(0)`. To prevent from accidentally depositing large amounts in the first deposit (and effectively burning LP shares), the function checks that all amounts being deposited are between `10 * _pullExistentials[i]` and a full token. Nevertheless, one full token might still have significant value for some tokens, e.g., WBTC or ETH.

6.39 Possible Optimization on _chargePerformanceFees

Design | Low | Version 5 | Code Corrected

The function `_chargePerformanceFees` in `ERC20RootVault` mints LP tokens to the `treasury` address as follows:

```
uint256 toMint;
if (hwmsD18 > 0) {
    toMint = FullMath.mulDiv(baseSupply, lpPriceD18 - hwmsD18, hwmsD18);
    toMint = FullMath.mulDiv(toMint, performanceFee, CommonLibrary.DENOMINATOR);
}
lpPriceHighWaterMarkD18 = lpPriceD18;
_mint(treasury, toMint);
```

The function would be more gas efficient if the minting is executed only for non-zero values, hence only minting when the `if-condition` is satisfied.

Code corrected:

In the updated code, the statement `_mint(...)` is moved inside the `if-block`, hence minting only non-zero amounts.

6.40 Possible Violation of the Minimum Token Amounts After the First Deposit

Design Low Version 5 Code Corrected

The function `ERC20RootVault.deposit` checks on the first deposit that all token amounts are larger than a minimum value `10 * _pullExistentials[i]`. If the TVL for a token goes below the threshold, users cannot make deposits for that token. However, the first depositor can circumvent the restriction for the minimum token amounts by performing an withdrawal after the deposit.

Code corrected:

The issue presented above is not present anymore in the updated code base as the first deposit always mints LP shares to `address(0)`.

6.41 Misleading Function Name and Natspec

Correctness Low Version 4 Code Corrected

The function `LStrategy.targetPrice` returns the price in x96 format. Neither the function name, nor the natspec description clarify the format of the return value. We have reported another issue in a calling function which assumed the price to be returned in a different format.

Code corrected:

The codebase has been updated to make more explicit in the function name and natspec description of `getTargetPriceX96` that the returned price is in x96 format. Similarly, other functions that return the price in x96 format are renamed accordingly.

6.42 Mismatch of Specifications for StrategyParams

Correctness Low Version 4 Code Corrected

The natspec description for the `struct StrategyParams` states that the params are changed with a delay:

```
/// @notice Params that could be changed by Strategy or Protocol Governance
    with Protocol Governance delay.
```

while the natspec description of the function `setStrategyParams` states that they are changed immediately, which is in line with the implementation:

```
// @notice Set Strategy params, i.e. Params that could be changed by Strategy or Protocol Governance immediately.
```

Core corrected

The natspec was corrected and does not mention the governance delay.

6.43 Missing Sanity Check for maxSlippageD in MStrategy

Design Low Version 4 Code Corrected

The function `MStrategy.setOracleParams` does not check that `maxSlippageD` is greater than zero, but if it is accidentally set to zero, the following code will revert always: .. code::solidity

require(absoluteDeviation < oracleParams.maxTickDeviation, ExceptionsLibrary.INVARIANT);

Code corrected:

The function `setOracleParams` is updated to include a check that the new `maxSlippageD` parameter is not zero:

```
require((params.maxSlippageD > 0) && (params.maxSlippageD <= DENOMINATOR), ExceptionsLibrary.INVARIANT);
```

6.44 Missing Sanity Checks for oracleSafetyMask

Design Low Version 4 Code Corrected

The function `LStrategy.updateTradingParams` performs sanity checks on the `maxSlippageD`, `orderDeadline` and `oracle`, but no checks are performed for `oracleSafetyMask`. This parameter should be non-zero for functions that query the oracle to work properly. Additionally, the function could check that at least one oracle with high safety index is included always.

Code corrected:

An additional check is added when new trading params are set by the admin. The check fort the new oracle safety mask is: `newTradingParams.oracleSafetyMask > 3`.

6.45 Possible Struct Optimization in Strategies

Design Low Version 4 Code Corrected

Mellow Finance might want to consider to optimize some structs in the code base. E.g., in:

```
struct TradingParams {
    uint32 maxSlippageD;
    uint32 orderDeadline;
    uint256 oracleSafetyMask;
    IOracle oracle;
    ...

struct PreOrder {
    address tokenIn;
    address tokenOut;
    uint256 amountIn;
```

```
    uint256 minAmountOut;
    uint256 deadline;
}

struct RatioParams {
    int24 tickMin;
    int24 tickMax;
    uint256 erc20MoneyRatioD;
    int24 minTickRebalanceThreshold;
    int24 tickNeighborhood;
    int24 tickIncrease;
    uint256 minErc20MoneyRatioDeviation0D;
    uint256 minErc20MoneyRatioDeviation1D;
}
```

Some of the variables will not take up a whole word and could be reordered to be packed tightly if needed.

Code corrected:

The variables in the structs listed above are reordered to be more efficient when stored in storage in the updated code.

6.46 Redundant Comparisons

Design | Low | Version 4 | Code Corrected

The function `Univ3Vault._getMinMaxPrice` implements the following code:

```
minPriceX96 = prices[0];
maxPriceX96 = prices[0];
for (uint32 i = 0; i < prices.length; ++i) {
    if (prices[i] < minPriceX96) {
        ...
```

Note that `minPriceX96` and `maxPriceX96` are assigned to `prices[0]` before the `for-loop`, so the first iteration of the loop is redundant.

Code corrected:

The `for-loop` has been updated to start from `i = 1` which avoids the redundant checks.

6.47 Redundant Storage Read in `ERC20Vault._pull`

Design | Low | Version 4 | Code Corrected

`_vaultTokens` is a state variable that is read multiple times in the `_pull` function even though it is stored in memory at the beginning of the function in `tokens`.

Code corrected:

The function has been revised to avoid storage reads for `_vaultTokens`, instead the value stored in memory `tokens` is now used.

6.48 Variables Can Be Declared as Constant

Design | Low | Version 4 | Code Corrected

The variable `MAX_ESTIMATED_AAVE_APY` in `AaveVaultGovernance` is declared as `immutable` and assigned to a constant in constructor. Similarly, `MAX_PROTOCOL_FEE`, `MAX_MANAGEMENT_FEE` and `MAX_PERFORMANCE_FEE` in `ERC20RootVaultGovernance` can be declared as constants.

Code corrected:

All `immutable` variables listed above are converted to constants.

6.49 Incorrect Specification for reclaimTokens

Correctness | Low | Version 3 | Specification Changed

The following statement in `IntegrationVault` regarding the function `reclaimTokens` is incorrect:

```
/// `reclaimTokens` for mistakenly transfered tokens (not included into vaultTokens)
/// additionally can be withdrawn by the protocol admin
```

Specification changed:

The statement in `IntegrationVault` has been changed as:

```
/// `reclaimTokens` for claiming rewards given by an underlying protocol to erc20Vault
     in order to sell them there
```

6.50 Missing Natspec Description for `minDeviation`

Correctness | Low | Version 2 | Code Corrected

The parameter `minDeviation` in the function `LStrategy._liquidityDelta` has no natspec description.

Code Corrected:

The description for `minDeviation` was added.

6.51 Casting of `maxTickDeviation`

Security Low Version 1 Code Corrected

`maxTickDeviation` is declared as `uint24` in the struct `OracleParams`. In function `_getAverageTickChecked`, the variable is casted as `int24`:

```
int24 maxDeviation = int24(oracleParams.maxTickDeviation);
```

For large values of `maxTickDeviation`, an overflow can happen when casting as `int24`.

Code corrected:

The deviation is now converted to an absolute value and directly compared to the `maxDeviation` without casting it to an `int24`.

6.52 Check Requirements First

Design Low Version 1 Code Corrected

Multiple functions can be more efficient by checking all requirements first (fail early), before performing expensive operations, such as external calls. We list below some examples (not an exhaustive list):

- `UniV2Validator`: in `validate` both branches of the `if` condition require the `msg.sender` to be the address `to`. The function can be optimized by checking the requirement first, and then performing the call to `_verifyPath` function.
- `UniV2Validator`: the function `_verifyPath` can be optimized by checking the following requirement first, before making external calls in the loop:

  ```
  require(vault.isVaultToken(path[path.length - 1]), ExceptionsLibrary.INVALID_TOKEN);
  ```

- `UniV3Validator`: the function `_verifyMultiCall` can be optimized by checking the following requirement first, before iterating through `path` and making external calls:

  ```
  require(recipient == address(vault), ExceptionsLibrary.INVALID_TARGET);
  ```

Code corrected:

The updated code Version 3 performs the checks first before executing other operations that might be expensive for the cases listed above.

6.53 Duplicate Code `_permissionIdsToMask`

Design Low Version 1 Code Corrected

The function `revokePermissions` in the `ProtocolGovernance` contract implements the following loop:

```
uint256 diff;
for (uint256 i = 0; i < permissionIds.length; ++i) {
        diff |= 1 << permissionIds[i];
}
```

which is a duplicate of the `_permissionIdsToMask` function.

Code corrected:

The code part was replaced by a call to the `_permissionIdsToMask` function.

6.54 Duplicate Storage Read in Deposit

Design | Low | Version 1 | Code Corrected

In `ERC20RootVault.deposit` the variable `totalSupply` is read for the check if it is *0* and later again to be loaded into memory.

Code corrected:

The redundant storage read is eliminated in the updated code and the value stored in memory `supply` is used instead.

6.55 Inconsistent Specifications

Correctness | Low | Version 1 | Specification Changed

In the specifications of struct `IProtocolGovernance.Params`:

- `permissionless` is described but it's not a member of the struct.
- `maxTokensPerVault` has the description that it stores the maximum tokens managed by the protocol, not a vault as the name suggests.
- `protocolTreasury` is not described.

In the specifications of `unitPrices`, the comment `staged for commit` is wrong.

Specifications changed:

The specifications have been updated in Version 2 to address the issues reported above.

6.56 Inefficient Array Shrinking

Design | Low | Version 1 | Code Corrected

`ProtocolGovernance.addressesByPermission` and `ProtocolGovernance.commitAllPermissionGrantsSurpassedDelay` create arrays with extended length and copy the values to a newly generated array with the correct size. This can be more efficiently done with `mstore` assembly, which is also used in various other places in the code.

Code corrected:

The array is now cut to length via `mstore` as in other parts of the code.

6.57 Inefficient State Variable Packing

Design | Low | Version 1 | Code Corrected

`lastFeeCharge` and `totalWithdrawnAmountsTimestamp` in `ERC20RootVault` are declared as `uint256`. Both are timestamps; hence, it might be more efficient to pack them as `uint64`. This only makes sense if they are used and loaded together, which would be possible in the current code base. Similarly, other structs in other contracts can be more storage-efficient by packing variables together.

Code corrected:

Both variables `lastFeeCharge` and `totalWithdrawnAmountsTimestamp` have been declared as `uint64` in the updated code.

6.58 Misleading Naming of Variables in `UniV3Oracle`

Design | Low | Version 1 | Code Corrected

The function `price` uses variable names that are inconsistent with the variable names of `Uniswap`. Namely, the variables `tick0` and `tick1` refer to `tickCumulative` variables of `Uniswap` and not normal ticks.

Similarly, the array `pricesX96` temporarily stores prices in square root format which are typically referred to as `sqrtPriceX96`. These inconsistencies make the reading of the code harder.

Code Corrected:

The variables were renamed accordingly.

6.59 Missing Sanity Check in `MStrategy.createStrategy`

Design | Low | Version 1 | Code Corrected

In `MStrategy.createStrategy` any token array could be passed in, but the strategy can only handle two tokens. There is no sanity check to limit the number of tokens. The fee parameter is also not checked even though it could only take a limited range of values.

Code corrected:

The sanity check on the tokens array is added in the `initialize` function which is called when a new strategy is created. The sanity check for the fee parameter is performed when the `pool` address is queried:

```
pool = IUniswapV3Pool(factory.getPool(tokens[0], tokens[1], fee_));
require(address(pool) != address(0), ExceptionsLibrary.ADDRESS_ZERO);
```

6.60 Missing Sanity Checks for Params

Design Low Version 1 Code Corrected

`LStrategy.updateRatioParams` and `LStrategy.updateOtherParams` do not perform sanity checks on all the params.

Code Corrected:

Both functions now perform basic sanity checks for the arguments.

6.61 Misspelled Variable Names

Design Low Version 1 Code Corrected

Function `deposit` in `ERC20RootVault` declares a variable with misspelled name: `delayedStaretgyParams`.

Struct `ratioParams` in `MStrategy` declares a variable with misspelled name: `tickNeiborhood`.

Code corrected:

Both variable names have been corrected in the updated code.

6.62 Possible Struct Optimization

Design Low Version 1 Code Corrected

Mellow Finance might want to consider to optimize some structs in the code base. E.g., in:

```
struct TradingParams {
    uint256 maxSlippageD;
    uint256 minRebalanceWaitTime;

    ...

struct RatioParams {
    uint256 erc20UniV3CapitalRatioD;
    uint256 erc20TokenRatioD;
    uint256 minErc20UniV3CapitalRatioDeviationD;
    uint256 minErc20TokenRatioDeviationD;
    uint256 minUniV3LiquidityRatioDeviationD
```

Some of the variables will not take up a whole word and could be packed if needed.

Code corrected:

The examples above and some other structs were changed. We assume that Mellow Finance evaluated all structs if an optimization is suitable and shall be applied.

6.63 Rebalance in `MStrategy` Is Inconsistent

Correctness | Low | Version 1 | Code Corrected

`MStrategy` provides only one function for rebalancing (`rebalance`) which calls `_rebalancePools` to enforce the predetermined ratio for the pools (`erc20Vault` and `moneyVault`) and then calls `_rebalanceTokens` to enforce the token ratio for the `erc20Vault`. The latter calls `_swapToTarget` which, in specific cases, pulls tokens from the `moneyVault` to the `erc20Vault`:

```
if (amountIn > erc20Tvl[tokenInIndex]) {
    ...
    moneyVault_.pull(address(erc20Vault_), tokens_, tokenAmounts, "");
    ...
}
```

This transfer of tokens from `moneyVault` to the `erc20Vault` would break the balance set in the function `_rebalancePools` called in the beginning of the rebalance process.

Code corrected:

The function `rebalance` has been updated to perform first the rebalance of tokens in the `erc20Vault`, which includes any potential swap. Afterwards, the function calls `_rebalancePools` which enforces the predetermined ratio of TVLs for the `erc20Vault` and `moneyVault`.

6.64 Specification for `minDeviation` Not Enforced

Correctness | Low | Version 1 | Code Corrected

The function `rebalanceERC20UniV3Vaults` in `LStrategy` calls the function `_liquidityDelta` and provides the minimum required deviation for a rebalance to be performed. `_liquidityDelta` checks the current deviation and if it is lower than the required minimum, it returns `0`. However, the calling function does not check the return value, hence continues the execution of the function although no tokens will be moved.

Code corrected:

The check below for the return value of the function `_liquidityDelta` has been added. Now the function returns immediately if `capitalDelta` is equal to 0 due to current deviation being smaller than the minimum required deviation:

```
(capitalDelta, isNegativeCapitalDelta) = _liquidityDelta(...);
if (capitalDelta == 0) {
    return (pulledAmounts, false);
}
```

6.65 Storing Redundant Data in Storage

Design Low Version 1 Code Corrected

The function `_addUniV3Pools` stores two entities in the mapping for each pair of tokens:

```
poolsIndex[token0][token1] = pool;
poolsIndex[token1][token0] = pool;
```

Given that there is only one Uniswap pool for a pair of tokens and a fee, the tokens can be sorted and stored only once in the mapping: `tokenA -> tokenB -> pool`, assuming `tokenA < tokenB`.

Code corrected:

The mapping `poolsIndex` now stores only one entry for a pair of tokens `token0 -> token1 -> pool`.

6.66 Unnecessary Approval to Vault Registry

Design Low Version 1 Code Corrected

Function `_initialize` in `Vault` has the following line which gives approval to the vault registry, but it is unnecessary as `VaultRegistry` is the implementation contract of the NFT token:

```
registry.setApprovalForAll(address(registry), true);
```

Code corrected:

The statement giving the approval has been removed from the function in Version 2.

6.67 Unused Constant in `ERC20Validator`

Design Low Version 1 Code Corrected

`ERC20Validator` declares the following constant, but it is not used:

```
bytes4 public constant EXCHANGE_SELECTOR = 0x3df02124;
```

Code corrected:

The constant was removed from the contract.

6.68 Unused Event `DeployedVault`

Design Low Version 1 Code Corrected

The contract `VaultGovernance` defines the event `DeployedVault` but it is not used in the current code base.

Code corrected:

The updated code emits the event `DeployedVault` when a new vault is created.

6.69 Unused Function `LStrategy._priceX96FromTick`

Design Low Version 1 Code Corrected

The internal function `LStrategy._priceX96FromTick` is not used in the LStrategy.

Code corrected:

The function was removed from the L Strategy.

6.70 Unused Imports

Design Low Version 1 Code Corrected

Throughout the code base we found many unused imports. Due to the number of unused imports, the following list is non-exhaustive and list only examples:

-`MellowOracle`

```
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol"
import "../libraries/CommonLibrary.sol";
```

- `UniV2Oracle`

```
import "../libraries/ExceptionsLibrary.sol"
```

- `UniV3Oracle`

```
import "../libraries/ExceptionsLibrary.sol"
```

- `LStrategy`

```
import "../interfaces/IVaultRegistry.sol"
import "../interfaces/utils/IContractMeta.sol"
```

- `MStrategy`

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

- `CowswapValidator`

```
import "../libraries/CommonLibrary.sol"
import "../libraries/PermissionIdsLibrary.sol"
```

- `CurveValidator`

```
import "../libraries/CommonLibrary.sol"
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol"
import "../interfaces/validators/IValidator.sol";
```

- `ERC20Validator`

```
import "../libraries/CommonLibrary.sol"
```

- `UniV2Validator` and `UniV3Validator`

```
import "../interfaces/validators/IValidator.sol";
import "../libraries/CommonLibrary.sol"
import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol"
```

- `AaveVault`

```
import "../interfaces/vaults/IVault.sol"
```

- `AggregateVault`

```
import "../interfaces/vaults/IAggregateVault.sol";
import "../libraries/PermissionIdsLibrary.sol"
```

- `ERC20RootVault`

```
import "../interfaces/utils/IContractMeta.sol"
```

Code partially corrected:

The unused imports have been removed from the respective contracts for all examples listed above, except for the `SafeERC20` import in the `MStrategy`.

6.71 Wrong Check of Minimum Token Amounts in `ERC20RootVault.withdraw`

Correctness | Low | Version 1 | Code Corrected

`ERC20RootVault.withdraw` compares the token amounts a user wants to receive at minimum with the calculated token amounts, but not the token amounts that are actually returned after pulling from underlying `Vault` s. This could potentially result in the user receiving less tokens than anticipated.

Code corrected:

The actual token amounts pulled from vaults are now validated against the minimum amounts provided by the user: `` require(actualTokenAmounts[i] >= minTokenAmounts[i],...);``

6.72 Wrong Specification for `YearnVault.tvl`

Correctness | Low | Version 1 | Specification Changed

The specification in `YearnVault` mentions that `YearnVault.tvl` returns a cached value when in fact it does not.

Specification changed:

The specification has been updated in Version 3 and the statement about the cached value has been removed.

6.73 `ContractRegistry` DOS

Security | Low | Version 1 | Code Corrected

`ContractRegistry.registerContract` checks that the version of a registered contract is always increasing in:

```
require(newContractVersion > _latestVersion(newContractName), ExceptionsLibrary.INVARIANT);
```

If a contract is deployed with a version set to max uint, this would be the last contract possible to add to the system. No contracts could be added afterwards.

Code corrected:

Mellow Finance introduced major and minor contract version. The 16 right most bytes are the minor version and the remaining bytes to the right the major version. A require ensures that with each call to `registerContract` the major version can only increase by 1 with `newContractVersionMajor - latestContractVersionMajor <= 1`.

6.74 `ERC20Vault._pull` Forces Push of Wrong Amount of Tokens

Correctness | Low | Version 1 | Code Corrected

In `ERC20Vault._pull`, if tokens are not pulled to the `ERC20RootVault`, the receiving `Vault` is forced to `push` the received tokens. The token amounts to be pushed are set in `actualTokenAmounts`, but this variable is never used. Instead `tokenAmounts` is used.

Code corrected:

The code has been corrected to push into the integration vault the amounts as stored in `actualTokenAmounts`.

6.75 `IntegrationVault._root` Does Not Check the NFT of the Root Vault

Correctness Low Version 1 Code Corrected

`IntegrationVault._root` tries to verify the initialization of a given `Vault` and its corresponding `ERC20RootVault` with the following code:

```
require(thisNft + thisOwnerNft != 0, ExceptionsLibrary.INIT);
```

If `thisNft` is set (greater than 0) and `thisOwnerNft` equals 0, no revert will happen. `_root` is called in `pull` only. `pull` already checks that the argument `thisNft` given to `_root` is not equal to 0 which renders the `require` useless.

Code corrected:

The statement was changed and checks each variable separately if it is zero in `(thisNft != 0) && (thisOwnerNft != 0)`.

6.76 `VaultGovernance.commitInternalParams` Does Not Delete Staged Parameters

Design Low Version 1 Code Corrected

`VaultGovernance.commitInternalParams` does not delete the `_stagedInternalParams` state variable.

Code corrected:

The state variable `_stagedInternalParams` is now deleted after it is applied.

6.77 `registry.ownerOf` Is Called Twice in `IntegrationVault.pull`

Design Low Version 1 Code Corrected

`registry.ownerOf` is called twice with the same value in `IntegrationVault.pull`, inducing unnecessary additional gas costs.

Code corrected:

The obvious redundant call to `registry.ownerOf` was removed. Still, there would be another call in `_isApprovedOrOwner`.

7 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.

7.1 Approximated TVL for Aave Vaults

Note Version 4

The function `AaveVault.tvl()` computes an approximate total value locked (TVL) based on the time passed since the last time the function `updateTvls` was called and the parameter `estimatedAaveAPY`:

```
uint256 apy = IAaveVaultGovernance(address(_vaultGovernance)).delayedProtocolParams().estimatedAaveAPY;
factor = CommonLibrary.DENOMINATOR + FullMath.mulDiv(apy, timeElapsed, CommonLibrary.YEAR);
```

Note that the parameter `estimatedAaveAPY` is set by the protocol admin for all tokens of the vault, hence the function `tvl` might return incorrect values if `updateTvls` is not called frequently.

7.2 Balances Are Drained Faster in Vaults With Lower Index

Note Version 1

`AggregateVault._pull` pulls funds out of the underlying Vault's by pulling the maximum amount out of each Vault sequentially. This drains funds faster from Vault's depending on their index in the `_subvaultNfts` state variable.

7.3 Deposits Can Be Blocked by Updating StrategyParams

Note Version 1

The function `ERC20RootVaultGovernance.setStrategyParams` does not perform any sanity check for the new parameters being set, hence if `tokenLimitPerAddress` or `tokenLimit` is set to zero, the functionality to deposit is blocked. The sanity checks are not enforced intentionally as the admin might use these parameters to block deposits into a root vault by updating these parameters.

7.4 Deprecated Function `_setupRole`

Note Version 1

`DefaultAccessControl` and `DefaultAccessControlLateInit` use the function `_setupRole`, which according to its specification is deprecated:

```
/**
 * NOTE: This function is deprecated in favor of {_grantRole}.
 */
```

7.5 Duplicate Declaration of `DENOMINATOR`

Note Version 1

Both `MStrategy` and `LStrategy` import `CommonLibrary` which declares the constant `DENOMINATOR`, however, they also declare the constant as well.

7.6 Dust LP Shares Are Burned

Note Version 6

If a user decides to redeem its LP shares in a root vault by calling the function `withdraw`, and if at the time of this action the amount of remaining LP shares represents less than the threshold `existentials` in underlying tokens, the whole user's LP balance is burned. Put shortly, the function prevents users from leaving dust amounts in LP shares when withdrawing.

7.7 External Functions in `ContractMeta`

Note Version 1

`ContractMeta` implements `external pure` functions, and currently they are called only by `registerContract` in `ContractRegistry`. The calls are performed as three external calls, which increase gas costs, as there is no function in `ContractMeta` returning all values in a single external call.

7.8 LP Tokens of the First Deposit Are Burned

Note Version 6

In Version 6 of the code base, the LP tokens of the first user depositing into a root vault are always send to `address(0)`, practically burning them.

7.9 Locked Token or ETH

Note Version 1

ERC20 tokens could be accidentally/intentionally sent to any contract. In such cases the tokens will be locked. Only `externalCall` for intergration vaults offers some functionality to recover funds.

7.10 No Checks for Address `to` on ERC20Token Transfer

Note Version 1

The functions `transfer` and `transferFrom` in `ERC20Token` do not perform any sanity check for the address `to`, hence making it possible to burn tokens by sending them to address `0x0`.

7.11 Non Canonical Signatures

Note Version 4

The function `IntegrationVault.isValidSignature` uses the library function `CommonLibrary.recoverSigner` to validate signatures if the strategy is an externally owned account. Note that, the function `recoverSigner` does not perform any sanity check on values `r`, `s` and `v` to ensure that only unique signatures validate successfully. Therefore, callers of this function should be aware of possible attacks (https://swcregistry.io/docs/SWC-117).

7.12 Non-indexed Event Topics

Note Version 1

Some events have already hit the limit of three indexed topics. But the events in `UnitPricesGovernance` have not and do not index the token address. Given that the unit price update could be important to users, making the token address `indexed`, makes it easier to filter the events for specific tokens.

There are some other events like `DeployedVault` in `VaultGovernance`, `ReclaimTokens` and `Pull` in `IntegrationVault` and `RebalancedUniV3` in `LStrategy` where one more index could be set. Additionally, some events could emit the nft which might be worth indexing (it e.g., is done in `SetStrategyParams`). This is just noted and up to Mellow Finance to decide.

7.13 OracleParams in MStrategy

Note Version 5

Te admin of `MStrategy` should carefully set the `OracleParams`. The admin should ensure that the Uniswap used for the oracle has enough observations to cover `oracleObservationDelta`, otherwise the function `_getAverageTickChecked` called in `_rebalanceTokens` will only use the spot price, hence making the rebalance function vulnerable to sandwich attacks. Additionally, the parameter `maxTickDeviation` should be carefully chosen to enforce proper slippage protection for the rebalance.

7.14 Performance Fee Capped

Note Version 5

`ERC20RootVault._chargePerformanceFees` only charges performance fees for the strategy if the price of LP tokens has reached a new high score. When prices have fallen, the fees are still not charged even when prices climb again until this all-time high has been reached again.

Additionally, if all liquidity providers withdraw their funds and the `totalSupply` is zero, or all token TVLs are less than `_pullExistentials`, the previous high score `lpPriceHighWaterMarkD18` is not reset, hence performance fees might not be collected as expected.

7.15 Rebalance of Uniswap Vaults in LStrategy

Note Version 4

The function `rebalanceUniV3Vaults` maintains a ratio of tokens in the two Uniswap positions depending on the move of the current price. If the price goes up, more tokens are transferred into the `upperVault` from the `lowerVault`, and vice-versa. The function is designed in a way that it tries to add

the same liquidity amount into the destination vault that is removed from the other vault. However, since the two vaults operate in different price ranges, the same liquidity amount translates into different token amounts. The token in the cash position (`erc20Vault`) are used to cover for the difference. Consequently the ratio between the cash position (`erc20Vault`) and the money vaults (`lowerVault` and `upperVault`) is affected.

7.16 Rollback Individual Validators Not Possible

Note Version 1

`ProtocolGovernance` implements a function to rollback all staged validators, but there is no functionality to rollback individual staged validators.

7.17 Special Behavior in `ERC20Token`

Note Version 1

The function `transferFrom` has a special behavior when `allowance==type(uint256).max`, as the allowance is never reduced when these transfers occur. This special behavior should be properly documented as users should be aware of it.

7.18 Trust Setup

Note Version 1

The system has multiple trusted roles and heavily relies on admin operations to work. E.g., setting oracles and the admin needs to maintain enough funds to open new Uniswap positions.

7.19 Uneven Gas Distribution on `deposit` and `withdraw`

Note Version 1

Fees are not calculated on every transaction. Therefore, some users are burdened with more gas costs than others depending on the time they are performing their `withdraw` and `deposit` actions.

7.20 Unit Prices Amounts

Note Version 5

The admin in `UnitPricesGovernance` can set the amounts of a given token that match the value of 1 USD. The prices are set with a delay of 14 days, hence the prices are not supposed to reflect the market price. Note that, for valuable tokens with few decimals, it might be impossible to store the correct token amount that matches 1 USD.

7.21 Unnecessary Creation of `Pair`

Note **Version 1**

In `UniV3Vault._push` and `UniV3Vault._pullUniV3Nft`, a `Pair` is created and not used as a `Pair` afterwards. Instead, the particular values are extracted from the `Pair`, rendering the creation of the `Pair` useless.

7.22 `ContractRegistry` Functions Truncate `name`

Note **Version 1**

Functions `versions`, `versionAddress` and `latestVersion` in `ContractRegistry` truncate the input parameter `name_` to 32 bytes:

```
bytes32 name = bytes32(bytes(name_));
```

If these functions are called with `name_` longer than 32 bytes, the return value would be based on the truncated input parameter `name_`, which is inconsistent behavior.

Furthermore, the function `latestVersion` parses the input parameter `name_` differently from other functions:

```
bytes32 name = bytes32(abi.encodePacked(name_));
```

7.23 `LStrategy` Needs Tokens to Create Uniswap Positions

Note **Version 1**

The function `_mintNewNft` assumes that the strategy contract has enough balance to open new Uniswap positions as needed, otherwise new Uniswap NFTs cannot be minted:

```
IERC20(tokens[0]).safeApprove(address(positionManager), minToken0ForOpening);
IERC20(tokens[1]).safeApprove(address(positionManager), minToken1ForOpening);
(newNft, , , ) = positionManager.mint(
    INonfungiblePositionManager.MintParams({
        token0: tokens[0],
        token1: tokens[1],
        fee: poolFee,
        tickLower: lowerTick,
        tickUpper: upperTick,
        amount0Desired: minToken0ForOpening, // required balance
        amount1Desired: minToken1ForOpening, // required balance
        amount0Min: 0,
        amount1Min: 0,
        recipient: address(this),
        deadline: deadline
    })
);
```

Mellow Finance is aware of this requirement and states they will take care that enough funds are available at any point in time. Additionally, a check was added to ensure that the amount of token needed in the contract is very low (less than `10**9`) to mitigate that money is lost because of the deactivated slippage protection in the function above.

7.24 `_pullExistentials` Are Unevenly Distributed in Terms of Value

Note **Version 1**

`_pullExistentials` in `AggregateVault` are set to `10**(token.decimals() / 2)` for each token. This is an uneven distribution considering that tokens may have different value. The existential for `USDT` for example has a much lower value than the existential for `WBTC`.

7.25 `addressesByPermission` Does Not Consider Forced Permissions

Note **Version 1**

The function `addressesByPermission` in the protocol governance contract returns only addresses that explicitly have the `permissionId` in the mapping `permissionMasks`. However, if the `permissionId` is enforced by `forceAllowMask`, then all addresses are assumed to have the permission.