BlockSec

Security Audit Report for Mellow Vaults

Date: August 4, 2022

Version: 1.3

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.08.05 16:48:50 +08'00'

Contents

Report Manifest

Item	Description
Client	Mellow
Target	Mellow Vaults

Version History

Version	Date	Description
1.0	June 29, 2022	First Release
1.1	July 26, 2022	Minor Change
1.2	July 27, 2022	Minor Change
1.3	August 4, 2022	Final Commit

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The Mellow project provides an open platform for liquidity providers to earn rewards (from their liquidities), and strategists to earn performance fees (by implementing active liquidity management strategies to manipulate the liquidities). On one side, according to the design of the protocol, the vaults of the lowest layer are used to hold liquidities and interact with the underlying earning protocols like AAVE, Yearn and Uniswap. Based on them, multiple vaults are combined together as a group, and a top-layer `ERC20RootVault` is used to provide liquidity management as well as vault management for the vault group. On the other side, the vault management access control is based on NFTs, i.e., each vault is assigned with an NFT and the ownership is given to the root vault, while the approval is given to a strategy. The funds are allowed to be moved between vaults and pushed/pulled to/from the underlying protocols by the strategists. However, they cannot be withdrawn from the system. Besides, the protocol implements some operations relying on the token prices, and it adopts some external price oracles to prevent potential price manipulation.

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values of the repo [1] during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Note that the final commit of this project [2] is ahead of the commits listed in the table.

Project		Commit SHA
Mellow	Version 1	f4a46879cbe2e7555df7ba33e6eb81dd7cfd4513
	Version 2	82126916cd3c7fc1a26f507d06b8c8755908a915
	Version 3	db5c15038e11277049b9c2471bbb5031d25eb0ab
	Version 4	d7b0a4d842e70cd1704f67fe1d676ec7f7b923c0
	Version 5	b4250761505742c211428986ecb4189ae2e402fc
	Version 6	8ca35ad8d24a14dea69f74790a7e5363fc87e2c2
	Version 7	165dfae8370d13227b5264d8c11200e16d426c6d
	Version 8	dd00a6a334b16a599f358dcc9b3de86f1aea959e
	Version 9	765376ae58677e4c7b5222a679bd4c9faa5c0da4
	Version 10	a463f3a0a14ca47ca5eef3a56c47ed3263853b2d
	Version 11	3caaf41d49e0cbcff9cfa5a5eefe2ae2478ceba8
	Version 12	314a9e0ac6e959a1278590198731d11a75af5af9

[1] https://github.com/mellow-finance/mellow-vaults

[2] The commit hash is ed3e07e5b873dbe6f4e5d632d0adc1f5b47dec8e.

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	High	Low
High	High	Medium
Low	Medium	Low

- **Fixed** The issue has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **nine** potential issues. We have **five** recommendations.

- High Risk: 0
- Medium Risk: 4
- Low Risk: 5
- Recommendations: 5

ID	Severity	Description	Category	Status
1	Low	Potential conflict of access control in `LStrategy`	Software Security	Fixed
2	Low	Unchecked governance parameters	Software Security	Fixed
3	Medium	Lack of checks on the vault type for `AggregateVault`	DeFi Security	Fixed
4	Low	Undetermined allocation for the liquidity mining rewards	DeFi Security	Acknowledged
5	Low	Potential dust tokens left in the vault	DeFi Security	Fixed
6	Medium	The delay mechanism to update the validator parameters could be disabled	DeFi Security	Fixed
7	Medium	Improper price calculation in the `_getTvlToken0` function	DeFi Security	Fixed
8	Low	Incorrect TVL calculation of the AAVE vault	DeFi Security	Fixed
9	Medium	Lack of access control for the new governance function	DeFi Security	Fixed
10	-	Remove unnecessary checks in ERC20RootVault	Recommendation	Acknowledged
11	-	Avoid using shadowed variables	Recommendation	Fixed
12	-	Use `mulDiv` to prevent precision losses	Recommendation	Fixed
13	-	Fix incorrect event variables	Recommendation	Fixed
14	-	Inconsistent slippage checks in `deposit` and `withdraw`	Recommendation	Acknowledged

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential conflict of access control in `LStrategy`

Severity Low

Status Fixed in `Version 12`

Introduced by `Version 1`

Description To rebalance the liquidities between the ERC20 vaults and Uniswap V3 vaults, the `deposit` and `withdraw` callback functions of the `ERC20RootVault` are implemented in the `LStrategy` contract. However, the function to perform the rebalancing (i.e., the `rebalanceERC20UniV3Vaults`) requires that the caller must be at least the `operator` role, i.e., an off-chain bot according to the assumption. As a result, it would require the ERC20RootVault to be set as the operator role, which might cause a conflict of the access control.

Impact N/A

Suggestion Add an additional role for the ERC20RootVault contract.

Feedback from the Project Since the ERC20RootVault is the owner of the LStrategy, it already has operator rights, therefore, the _requireAtLeastOperator check is not superfluous here.

Update: We are now evaluating two scenarios: either make rebalances available for the public (hence, we will have to add on-chain slippage protection onto our contacts) or remove callbacks, both solutions solve the problem of inconsistent access control. The choice hasn't been made yet and we're currently evaluating this issue.

2.1.2 Unchecked governance parameters

Severity Low

Status Fixed in Version 6

Introduced by Version 1

Description The Mellow project has a layered governance system. There is a protocol-wise governance contract named ProtocolGovernance which is used to set the global parameters and provide some global methods. Besides, for each vault, there exists a governance contract that sets various vault-specific parameters. For example, in AAVE vault governance contract, the address of the AAVE Lending Pool contract can be specified. However, in all governance contracts , there does not exist any check to verify the validity of the provided parameters, e.g., whether the address of the AAVE Lending Pool contract is a non-zero value.

Impact Invalid governance parameters may lead to unexpected behaviors.

Suggestion Add the sanity checks to verify the governance parameters.

2.2 DeFi Security

2.2.1 Lack of checks on the vault type for AggregateVault

Severity Medium

Status Fixed in Version 11

Introduced by Version 1

Description There is a special design of the vault system that the first vault (i.e., the vault with index 0, also referred as *zero vault*) is a special vault of the ERC20Vault type with special logic in the push and pull functions. However, in the _initialize function of the AggregateVault contract, there is no check to verify whether the *zero vault* is of the ERC20Vault type or not.

The design is based on an assumption that vaults are created and controlled by the project. However, to be able to serve as a permissionless vault system (i.e., allowing anyone to create vaults and strategies), this check must be specified to ensure the logical soundness of the AggregateVault system.

Impact The incorrect type of the *zero vault* may lead to improper handling of vault tokens.

Suggestion Add proper sanity checks.

2.2.2 Undetermined allocation for the liquidity mining rewards

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description The vaults in the Mellow project are managed to provide liquidity into the underlying projects to make profits. These projects may issue protocol tokens as the liquidity provision rewards. As a result, there should exist some logic to handle those protocol tokens in the vault contracts. However, this part of logic has not been implemented yet.

Specifically, the liquidity mining rewards could be claimed through the `externalCall` interface of the vaults [1]. However, invocations through this interface must be checked by some validator contracts, while no logic in the current validator contracts explicitly supports claiming rewards (except the validator that accepts any call). For example, the `MStrategy` contract that may deposit liquidity to AAVE or Yearn does not implement the corresponding logic to claim the liquidity mining rewards.

Impact Liquidity providers may suffer from losses because the liquidity mining rewards provided by some projects might not be claimed.

Suggestion N/A

Feedback from the Project For now, there is no such strategy that should claim rewards from internal protocols. Actually, `reclaimTokens` and `externalCall` methods propose a way of claiming reward tokens. The pipeline is the following: strategy claims reward tokens with `reclaimTokens` method. Then it uses `externalCall` method to swap it on ERC20Vault tokens. We expect such logic to be implemented in the strategy. If needed, it's possible to approve `AllowAllValidator` to make any needed external call from a strategy.

2.2.3 Potential dust tokens left in the vault

Severity Low

Status Fixed in `Version 7`

Introduced by `Version 1`

Description The `reclaimTokens` function in the `IntegrationVault` (i.e., the parent contract of the lowest level vaults) is used to reclaim tokens that are sent to the vault contracts by mistake. The tokens are transferred to the *zero vault* and can be claimed by invoking some external calls in the *zero vault*.

However, as shown in the following code snippet, if the reclaimed tokens are the `vaultToken`, the token amounts will be checked against the predefined small values stored in `_pullExistentials`, which may leave some dust of the tokens in the vaults.

```
function reclaimTokens(address[] memory tokens)
  external
  virtual
  nonReentrant
  returns (uint256[] memory actualTokenAmounts)
{
```

[1]Besides, some liquidity mining rewards are directly distributed to the LP address (i.e., the vaults interact with the underlying protocol). However, these rewards can only be claimed to the *zero vault* and cannot be withdrawn.

```
    uint256 nft_ = _nft;
    require(nft_ != 0, ExceptionsLibrary.INIT);
    IVaultGovernance.InternalParams memory params = _vaultGovernance.internalParams();
    IProtocolGovernance governance = params.protocolGovernance;
    IVaultRegistry registry = params.registry;
    address owner = registry.ownerOf(nft_);
    address to = _root(registry, nft_, owner).subvaultAt(0);
    require(to != address(this), ExceptionsLibrary.INVARIANT);
    actualTokenAmounts = new uint256[](tokens.length);
    for (uint256 i = 0; i < tokens.length; ++i) {
        require(
            governance.hasPermission(tokens[i], PermissionIdsLibrary.ERC20_TRANSFER),
            ExceptionsLibrary.INVALID_TOKEN
        );
        IERC20 token = IERC20(tokens[i]);
        actualTokenAmounts[i] = token.balanceOf(address(this));
        int256 vaultTokenIndex = getVaultTokenIndex(tokens[i]);
        if ((vaultTokenIndex != -1) && (actualTokenAmounts[i] <= _pullExistentials[uint256(vaultTokenIndex)]))
            continue;

        token.safeTransfer(to, actualTokenAmounts[i]);
    }
    emit ReclaimTokens(to, tokens, actualTokenAmounts);
}
```

Listing 2.1: IntegrationVault.sol

Impact Some dust of tokens may be left in the vault.

Suggestion N/A

2.2.4 The delay mechanism to update the validator parameters could be disabled

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 1`

Description A validator parameter can be updated through two steps, i.e., first staging the parameter and then committing it. The `BaseValidator` would add a delay in between the two steps. However, there exists an invocation path that the admin can set the parameter without any delay. Specifically, in the `commitValidatorParams` function, the variable named `_stagedValidatorParamsTimestamp` is used to check whether the timestamp exceeds the delay time or not, and it would be deleted after the check. Note that the deletion operation will set the variable to 0, hence the check in line 50 becomes `require(block.timestamp >= 0)` and can always be satisfied and bypassed. As a result, the admin can invoke the `commitValidatorParams` function again and set `_validatorParams` to 0, i.e., no delay anymore.

```
47 function commitValidatorParams() external {
48     IProtocolGovernance governance = _validatorParams.protocolGovernance;
49     require(governance.isAdmin(msg.sender), ExceptionsLibrary.FORBIDDEN);
50     require(block.timestamp >= _stagedValidatorParamsTimestamp, ExceptionsLibrary.TIMESTAMP);
51     _validatorParams = _stagedValidatorParams;
```

```
    delete _stagedValidatorParams;
    delete _stagedValidatorParamsTimestamp;
    emit CommittedValidatorParams(tx.origin, msg.sender, _validatorParams);
}
```

Listing 2.2: BaseValidator.sol

Impact The `commitValidatorParams` function can be called repeatedly to disable the delay, which may break the governance management.

Suggestion Check and set `_stagedValidatorParamsTimestamp` properly.

2.2.5 Improper price calculation in the `_getTvlToken0` function

Severity Medium

Status Fixed in `Version 10`

Introduced by `Version 1`

Description The `_getTvlToken0` function is used to calculate the total TVL. Specifically, the protocol fee will be calculated based on the first token (i.e., `token0`). However, the prices returned by the oracle will be scaled up by 2^{96}, which is not properly handled by the current implementation.

```
function _getTvlToken0(
    uint256[] memory tvls,
    address[] memory tokens,
    IOracle oracle
) internal view returns (uint256 tvl0) {
    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;
    }
}
```

Listing 2.3: ERC20RootVault.sol

Impact The protocol fee would be incorrectly calculated because the price is not improperly handled.

Suggestion Fix the calculation.

2.2.6 Incorrect TVL calculation of the AAVE vault

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `tvl` function of the AAVE vault, the TVL is represented as two arrays, i.e., `minTokenAmounts` and `maxTokenAmounts`, respectively. Specifically, `minTokenAmounts` comes from a state variable named `_tvls`, while `maxTokenAmounts` is calculated based on `minTokenAmounts` and an estimated APY which relies on the state variable named `_lastTvlUpdateTimestamp` (i.e., the last update timestamp of the TVL).

```
function tvl() public view override returns (uint256[] memory minTokenAmounts, uint256[] memory maxTokenAmounts) {
  minTokenAmounts = _tvls;
  maxTokenAmounts = new uint256[](minTokenAmounts.length);
  uint256 timeElapsed = block.timestamp - _lastTvlUpdateTimestamp;
  uint256 factor = CommonLibrary.DENOMINATOR;
  if (timeElapsed > 0) {
      uint256 apy = IAaveVaultGovernance(address(_vaultGovernance)).delayedProtocolParams().estimatedAaveAPY;
      factor = CommonLibrary.DENOMINATOR + FullMath.mulDiv(apy, timeElapsed, CommonLibrary.YEAR);
  }
  for (uint256 i = 0; i < minTokenAmounts.length; i++) {
      maxTokenAmounts[i] = FullMath.mulDiv(factor, minTokenAmounts[i], CommonLibrary.DENOMINATOR);
  }
}
```

Listing 2.4: AaveVault.sol

`_tvls` is updated as the balances of the deposited `aTokens` in the `updateTvls` function.

```
function updateTvls() external {
    _updateTvls();
}
```

Listing 2.5: AaveVault.sol

```
function _updateTvls() private {
    uint256 tvlsLength = _tvls.length;
    for (uint256 i = 0; i < tvlsLength; ++i) {
        _tvls[i] = IERC20(_aTokens[i]).balanceOf(address(this));
    }
}
```

Listing 2.6: AaveVault.sol

However, `_lastTvlUpdateTimestamp` is not properly updated, which may cause an incorrect calculation of the TVL of the AAVE vault.

Impact The TVL of the AAVE vault will be incorrectly calculated.

Suggestion Update `_lastTvlUpdateTimestamp` in the `_updateTvls` function.

2.2.7 Lack of access control for the new governance function

Severity Medium

Status Fixed in `Version 8`

Introduced by `Version 4`

Description In Version 4 a new function named commitYTokens is introduced to the governance contract of the Yearn vault. Once this function is called, no new mapping (from a token to the corresponding yToken) is allowed to set. However, this function does not have any access control, which means it could be arbitrarily invoked to affect all newly deployed governance contracts.

```
function commitYTokens() external {
   _tokensCommited = true;
}
```

Listing 2.7: YearnVaultGovernance.sol

Impact All newly deployed governance contracts of Yearn vaults might be affected due to the new governance function without any access control.

Suggestion Add proper access control for this function.

2.3 Additional Recommendation

2.3.1 Remove unnecessary checks in ERC20RootVault

Status Acknowledged

Introduced by Version 1

Description In the following code snippet, the check in line 254 is unnecessary, i.e., the inequality will not hold for all use cases.

```
241 function _getNormalizedAmount(
242     uint256 tvl_,
243     uint256 amount,
244     uint256 lpAmount,
245     uint256 supply
246 ) internal pure returns (uint256) {
247     if (supply == 0) {
248         // skip normalization on init
249         return amount;
250     }
251
252     // normalize amount
253     uint256 res = FullMath.mulDiv(tvl_, lpAmount, supply);
254     if (res > amount) {
255         res = amount;
256     }
257
258     return res;
259 }
```

Listing 2.8: ERC20RootVault.sol

Impact N/A

Suggestion Remove unnecessary checks to save gas.

Feedback from the Project In our opinion, the code looks more readable if this check remains.

2.3.2 Avoid using shadowed variables

Status Fixed in `Version 9`

Introduced by `Version 1`

Description In the `_initialize` function of the `AggregateVault` contract, there is a variable named `vaultTokens` that shadows a state variable with the same name.

```
function _initialize(
  address[] memory vaultTokens_,
  uint256 nft_,
  address strategy_,
  uint256[] memory subvaultNfts_
) internal virtual {
    IVaultRegistry vaultRegistry = IVaultGovernance(msg.sender).internalParams().registry;
    require(subvaultNfts_.length > 0, ExceptionsLibrary.EMPTY_LIST);
    for (uint256 i = 0; i < subvaultNfts_.length; i++) {
        uint256 subvaultNft = subvaultNfts_[i];
        require(subvaultNft > 0, ExceptionsLibrary.VALUE_ZERO);
        require(vaultRegistry.ownerOf(subvaultNft) == address(this), ExceptionsLibrary.FORBIDDEN);
        require(_subvaultNftsIndex[subvaultNft] == 0, ExceptionsLibrary.DUPLICATE);
        address vault = vaultRegistry.vaultForNft(subvaultNft);
        require(vault != address(0), ExceptionsLibrary.ADDRESS_ZERO);
        require(
            IIntegrationVault(vault).supportsInterface(type(IIntegrationVault).interfaceId),
            ExceptionsLibrary.INVALID_INTERFACE
        );
        address[] memory vaultTokens = IIntegrationVault(vault).vaultTokens();
        require(vaultTokens_.length == vaultTokens.length, ExceptionsLibrary.INVALID_LENGTH);
```

Listing 2.9: AggregateVault.sol

Impact N/A

Suggestion Rename one of the two variables.

2.3.3 Use `mulDiv` to prevent precision losses

Status Fixed in `Version 4`

Introduced by `Version 1`

Description The contracts have a mixed use of regular arithmetic expressions and the `FullMath` library. Some calculations can be rewritten in the form of `mulDiv` calls to prevent precision losses, including the following functions:

1. `_getTvlToken0` of the `ERC20RootVault` contract.
2. `tvl` of the `YearnVault` contract.

Impact N/A

Suggestion Rewrite these calculations.

2.3.4 Fix incorrect event variables

Status Fixed in `Version 5`

Introduced by `Version 1`

Description In the following code snippet, the addresses logged into the `added` variable are always zero addresses. Besides, the state variable `_pools` is never used.

```
function _addUniV3Pools(IUniswapV3Pool[] memory pools) internal {
    IUniswapV3Pool[] memory replaced = new IUniswapV3Pool[](pools.length);
    IUniswapV3Pool[] memory added = new IUniswapV3Pool[](pools.length);
    uint256 j;
    uint256 k;
    for (uint256 i = 0; i < pools.length; i++) {
        IUniswapV3Pool pool = pools[i];
        address token0 = pool.token0();
        address token1 = pool.token1();
        _pools.add(address(pool));
        IUniswapV3Pool currentPool = poolsIndex[token0][token1];
        if (address(currentPool) != address(0)) {
            replaced[j] = currentPool;
            j += 1;
        } else {
            added[k] = currentPool;
            k += 1;
        }
```

Listing 2.10: UniV3Oracle.sol

Impact N/A

Suggestion Fix the code accordingly.

2.3.5 Inconsistent slippage checks in `deposit` and `withdraw`

Status Acknowledged

Introduced by `Version 1`

Description The `ERC20RootVault` contract is used to handle the liquidity management, i.e., users deposit liquidity to get LP tokens, and burn LP tokens to retrieve liquidity back. Specifically, the `withdraw` function of the `ERC20RootVault` contract will pull from underlying vaults and check whether the actual pulled amount is larger than the `minTokenAmounts` (which serves as a slippage check). However, the `actualTokenAmounts` may not be the same as the `tokenAmounts` calculated before and there does not exist a second check to verify the actual token amounts regarding to the burnt LP token amounts.

Furthermore, in the `deposit` function, the token amounts deposited is derived from the real amounts pushed to the sub-vaults for minting LP tokens. The implementation is slightly inconsistent between the `deposit` and `withdraw` functions.

```
function withdraw(
  address to,
  uint256 lpTokenAmount,
  uint256[] memory minTokenAmounts,
```

```
    bytes[] memory vaultsOptions
) external nonReentrant returns (uint256[] memory actualTokenAmounts) {
    uint256 supply = totalSupply;
    require(supply > 0, ExceptionsLibrary.VALUE_ZERO);
    address[] memory tokens = _vaultTokens;
    uint256[] memory tokenAmounts = new uint256[](_vaultTokens.length);
    (uint256[] memory minTvl, ) = tvl();
    if (lpTokenAmount > balanceOf[msg.sender]) {
        lpTokenAmount = balanceOf[msg.sender];
    }
    for (uint256 i = 0; i < _vaultTokens.length; ++i) {
        tokenAmounts[i] = FullMath.mulDiv(lpTokenAmount, minTvl[i], supply);
    }
    actualTokenAmounts = _pull(address(this), tokenAmounts, vaultsOptions);
    for (uint256 i = 0; i < _vaultTokens.length; ++i) {
        require(actualTokenAmounts[i] >= minTokenAmounts[i], ExceptionsLibrary.LIMIT_UNDERFLOW);
    }
```

Listing 2.11: ERC20RootVault.sol

```
function deposit(
    uint256[] memory tokenAmounts,
    uint256 minLpTokens,
    bytes memory vaultOptions
) external nonReentrant returns (uint256[] memory actualTokenAmounts) {
    require(
        !IERC20RootVaultGovernance(address(_vaultGovernance)).operatorParams().disableDeposit,
        ExceptionsLibrary.FORBIDDEN
    );
    address[] memory tokens = _vaultTokens;
    if (totalSupply == 0) {
        for (uint256 i = 0; i < tokens.length; ++i) {
            require(tokenAmounts[i] > FIRST_DEPOSIT_LIMIT, ExceptionsLibrary.LIMIT_UNDERFLOW);
        }
    }
    (uint256[] memory minTvl, uint256[] memory maxTvl) = tvl();
    uint256 thisNft = _nft;
    IERC20RootVaultGovernance.DelayedStrategyParams memory delayedStrategyParams = IERC20RootVaultGovernance(
        address(_vaultGovernance)
    ).delayedStrategyParams(thisNft);
    require(
        !delayedStrategyParams.privateVault || _depositorsAllowlist.contains(msg.sender),
        ExceptionsLibrary.FORBIDDEN
    );
    uint256 supply = totalSupply;
    uint256 preLpAmount = _getLpAmount(maxTvl, tokenAmounts, supply);
    uint256[] memory normalizedAmounts = new uint256[](tokenAmounts.length);
    for (uint256 i = 0; i < tokens.length; ++i) {
        normalizedAmounts[i] = _getNormalizedAmount(maxTvl[i], tokenAmounts[i], preLpAmount, supply);
        IERC20(tokens[i]).safeTransferFrom(msg.sender, address(this), normalizedAmounts[i]);
    }
```

```
        actualTokenAmounts = _push(normalizedAmounts, vaultOptions);
        uint256 lpAmount = _getLpAmount(maxTvl, actualTokenAmounts, supply);
        require(lpAmount >= minLpTokens, ExceptionsLibrary.LIMIT_UNDERFLOW);
        require(lpAmount != 0, ExceptionsLibrary.VALUE_ZERO);
        IERC20RootVaultGovernance.StrategyParams memory params = IERC20RootVaultGovernance(address(
            _vaultGovernance))
            .strategyParams(thisNft);
        require(lpAmount + balanceOf[msg.sender] <= params.tokenLimitPerAddress, ExceptionsLibrary.
            LIMIT_OVERFLOW);
        require(lpAmount + totalSupply <= params.tokenLimit, ExceptionsLibrary.LIMIT_OVERFLOW);

        _chargeFees(thisNft, minTvl, supply, actualTokenAmounts, lpAmount, tokens, false);
        _mint(msg.sender, lpAmount);

        for (uint256 i = 0; i < _vaultTokens.length; ++i) {
            if (normalizedAmounts[i] > actualTokenAmounts[i]) {
                IERC20(_vaultTokens[i]).safeTransfer(msg.sender, normalizedAmounts[i] -
                    actualTokenAmounts[i]);
            }
        }

        if (delayedStrategyParams.depositCallbackAddress != address(0)) {
            ILpCallback(delayedStrategyParams.depositCallbackAddress).depositCallback();
        }

        emit Deposit(msg.sender, _vaultTokens, actualTokenAmounts, lpAmount);
    }
```

Listing 2.12: ERC20RootVault.sol

Impact The liquidity providers may suffer from losses.

Suggestion N/A

Feedback from the Project The `withdraw` function logic is that for the set `lpTokenAmount` and `minTokenAmonts` limits, the user will receive at least `minTokenAmonts` tokens, for `lpTokenAmount` lp-tokens. We think that additionally checking the number of burned lp tokens is redundant in this case.

2.4 Note

2.4.1 Strategy contracts must implement price slippage checks

Status Acknowledged

Introduced by `Version 1`

Description The `externalCall` interface allows the strategies to implement arbitrary calls on behalf of the vaults (i.e., the calls will be initiated from the vaults). Besides the normal token actions (e.g., `transfer` and `approve`), these calls can also be used to swap tokens between the vault and DEXes (e.g., Uniswap and Cowswap). There are some special contracts (i.e., the validators) to verify these calls. However, only the tokens and the recipient are checked in the validator contracts. Hence the unchecked price slippage may lead to token swaps in imbalanced or manipulatable pools, and eventually undermine the corresponding strategies. So either the Mellow core contracts or the strategy contracts must implement proper price slippage checks.

Impact The liquidity providers may suffer from losses due to the potential price slippage.

Suggestion It is important to enforce the price slippage checks. Specifically, if the strategies must be audited and whitelisted by the project administrators, they can still possibly be manipulated to swap in an imbalanced pool which may inevitably cause LPs to lose their funds. Besides, the strategists may enlist some scam strategies to attract the users and then rugpull through unchecked swaps using the `externalCall` interface.

Feedback from the Project The main idea behind validators is to check if arguments are valid. Our consideration here is the following: we will publish only audited and trusted strategies on our site. If the strategy is not audited and trusted by Mellow team, the strategist should persuade liquidity providers to put their liquidity by himself.