SMART CONTRACT AUDIT REPORT

for

MSTABLE-ICSMM

Prepared By: Shuxiao Wang

PeckShield
February 11, 2021

Document Properties

Client	mStable-ICSMM
Title	Smart Contract Audit Report
Target	mStable-ICSMM
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 11, 2021	Xuxian Jiang	Final Release
1.0-rc1	February 7, 2021	Xuxian Jiang	Release Candidate #1
0.3	January 31, 2021	Xuxian Jiang	Additional Findings #2
0.2	January 22, 2021	Xuxian Jiang	Additional Findings #1
0.1	January 20, 2021	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the mStable-ICSMM protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About mStable-ICSMM

The mStable-ICSMM protocol features the first ever implementation of a new AMM design, i.e., Incentivized Constant Sum Market Maker (abbreviated as ICSMM). It allows for the support of highly efficient, low slippage swaps with the enhanced protections through the new upper and lower bounds. Specifically, the mStable assets (or mAssets) are backed by a basket of assets (bAssets) of the same peg. These bAssets can be connected into lending markets, and can be swapped to produce swap fees. These system yields are redirected to a savings contract. The users have the option of receiving the system yield by depositing into the savings contract while the new design ensures mAssets maintain the necessary peg.

Table 1.1: Basic Information of mStable-ICSMM

Item	Description
Issuer	mStable-ICSMM
Website	https://mstable.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 11, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. It should be noted that this is not an economic audit and the correctness/reasoning of the underlying invariant is not part of this audit. In addition, while this repository contains a number of sub-directories, this audit covers only the `masset` sub-directory.

- https://github.com/mstable/mStable-ICSMM.git (0673e33)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/mstable/mStable-ICSMM.git (754e7db)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;

- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the mStable-ICSMM. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	6	
Informational	1	
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of mStable-ICSMM

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Improvements of CurvedMasset::initialize()	Business Logic	Fixed
PVE-002	Low	Early Break of Manager::negateIsolation()	Coding Practices	Fixed
PVE-003	Low	Constant Risk Parameters: forgeValidatorLocked/basket.failed	Business Logic	Confirmed
PVE-004	Low	Same Function Name With Different Definitions	Coding Practices	Fixed
PVE-005	Low	Removal Of Unused Code/Events	Coding Practices	Fixed
PVE-006	Low	Improved Corner Cases in Root()	Coding Practices	Fixed
PVE-007	Medium	Potential State Inconsistency From Memory Cache	Coding Practices	Fixed
PVE-008	Informational	Mixed Risk Parameters Between swapFee And redemptionFee	Business Logic	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Suggested Improvements of CurvedMasset::initialize()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CurvedMasset`
- Category: Business Logic [5]
- CWE subcategory: N/A

Description

The mStable-ICSMM implementation takes a proxy-based approach where the proxy contract is deployed at the front-end while the logic contract contains the actual business logic implementation. This approach has the flexible support in terms of upgradeability. However, the upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts.

Due to the inherent requirement of any proxy-based upgradeability system, no constructors may be used in upgradeable contracts. This means we need to move the constructor functionality of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

To elaborate, we show below the `initialize()` routine in the logic contract implementation, i.e., `CurvedMasset`. This routine is designed to set up the underlying assets (`bAsset`) and the invariant validator (`forgeValidator`) as well as initialize various risk parameters (e.g., `maxBassets`, `swapFee`, and `cacheSize`).

```
    /**
     * @dev Initialization function for upgradable proxy contract.
     *      This function should be called via Proxy just after contract deployment.
     *      To avoid variable shadowing appended 'Arg' after arguments name.
     * @param _nameArg          Name of the mAsset
     * @param _symbolArg        Symbol of the mAsset
     * @param _forgeValidator   Address of the AMM implementation
     * @param _bAssets          Array of erc20 bAsset addresses
```

```
     * @param _integrators      Matching array of the platform intergations for bAssets
     */
    function initialize(
        string calldata _nameArg,
        string calldata _symbolArg,
        address _forgeValidator,
        address[] calldata _bAssets,
        address[] calldata _integrators,
        bool[] calldata _hasTxFees
    ) public initializer {
        InitializableToken._initialize(_nameArg, _symbolArg);
        _initializeReentrancyGuard();

        forgeValidator = IInvariantValidator(_forgeValidator);

        require(_bAssets.length > 0, "No bAssets");

        maxBassets = 10;

        for (uint256 i = 0; i < _bAssets.length; i++) {
            Manager.addBasset(
                bAssetPersonal,
                bAssetData,
                bAssetIndexes,
                maxBassets,
                _bAssets[i],
                _integrators[i],
                1e8,
                _hasTxFees[i]
            );
        }

        MAX_FEE = 2e16;
        swapFee = 6e14;
        cacheSize = 1e17;
    }
```

Listing 3.1: CurvedMasset:: initialize ()

By examining the logic, we identify the following opportunities for further improvements:

- Parameter Validation: It is always preferred to perform a sanity check on given arguments. Especially, the `initialize()` routine can be benefited from verifying the given arguments, i.e., `_bAssets`, `_integrators`, and `_hasTxFees`, have the same length.
- maxBassets Adjustment: After taking the given array of underlying `bAssets` and populating them in the internal records (`bAssetPersonal` and `bAssetData`), the system-wide risk parameter of `maxBassets` is now known and can be fixed with the exact number of current `bAssets`. In other words, with the previous deduplicate check, we can compute it as follows: `maxBassets = _bAssets.length`.

Recommendation Validate the given arguments in `CurvedMasset::initialize()` and apply the above three improvements.

Status The issue has been fixed by this commit: `754e7db`.

3.2 Early Break of Manager::negateIsolation()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Manager`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

In the mStable-ICSMM protocol, there is a `Manager` contract that is designed to perform the `Basket` management duties for an `mAsset`. Also, for practical reason, it allows the logic to be abstracted to avoid bytecode inflation during deployment.

In our analysis, we notice there is a function named `negateIsolation()` that negates the isolation of a given `bAsset`. It implements a rather straightforward logic by firstly locating the current index of the given `bAsset`, next iterating the internal record of the `bAssetPersonal` array for its state, and finally updating the `Basket`-related state of `undergoingRecol`.

```
    /**
     * @dev Negates the isolation of a given bAsset
     * @param _basket           Struct containing core basket info
     * @param _bAssetPersonal  Basset data storage array
     * @param _bAssetIndexes    Mapping of bAsset address to their index
     * @param _bAsset Address of the bAsset
     */
    function negateIsolation(
        MassetStructs.BasketState storage _basket,
        MassetStructs.BassetPersonal[] storage _bAssetPersonal,
        mapping(address => uint8) storage _bAssetIndexes,
        address _bAsset
    ) external {
        uint256 i = _getAsset(_bAssetPersonal, _bAssetIndexes, _bAsset);

        _bAssetPersonal[i].status = MassetStructs.BassetStatus.Normal;

        bool undergoingRecol = false;
        for (uint256 j = 0; j < _bAssetPersonal.length; j++) {
            if (_bAssetPersonal[j].status != MassetStructs.BassetStatus.Normal) {
                undergoingRecol = true;
            }
        }
```

```
305         _basket.undergoingRecol = undergoingRecol;
306
307         emit BassetStatusChanged(_bAsset, MassetStructs.BassetStatus.Normal);
308     }
```

Listing 3.2: Manager:: negateIsolation

To elaborate, we show above the implementation of `negateIsolation()`. It comes to our attention that once there is a hit in the internal iteration (lines 300 – 304), we can simply break out of the internal `for`-loop without the need of finishing up the rest items in the array.

Recommendation Revise the `negateIsolation()` logic to make a early break in the internal `for`-loop.

Status The issue has been fixed by this commit: `754e7db`.

3.3 Constant Risk Parameters: forgeValidatorLocked/basket.failed

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CurvedMasset`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The mStable-ICSMM protocol is no exception. Specifically, if we examine the `CurvedMasset` contract, it has defined a number of system-wide risk parameters: `swapFee`, `cacheSize`, and `hasTxFee`. In the following, we show corresponding routines that allow for their changes.

```
    /**
     * @dev Upgrades the version of ForgeValidator protocol. Governor can do this
     *      only while ForgeValidator is unlocked.
     * @param _newForgeValidator Address of the new ForgeValidator
     */
    function upgradeForgeValidator(address _newForgeValidator) external override
        onlyGovernor {
        require(!forgeValidatorLocked, "Must be allowed to upgrade");
        require(_newForgeValidator != address(0), "Must be non null address");

        forgeValidator = IInvariantValidator(_newForgeValidator);

        emit ForgeValidatorChanged(_newForgeValidator);
```

```
    }

    /**
     * @dev Set the ecosystem fee for redeeming a mAsset
     * @param _swapFee Fee calculated in (%/100 * 1e18)
     */
    function setSwapFee(uint256 _swapFee) external override onlyGovernor {
        require(_swapFee <= MAX_FEE, "Rate must be within bounds");

        swapFee = _swapFee;

        emit SwapFeeChanged(_swapFee);
    }

    /**
     * @dev Update transfer fee flag for a given bAsset, should it change its fee
         practice
     * @param _bAsset   bAsset address
     * @param _flag         Charge transfer fee when its set to 'true', otherwise 'false
         '
     */
    function setTransferFeesFlag(address _bAsset, bool _flag) external override
        managerOrGovernor {
        Manager.setTransferFeesFlag(bAssetPersonal, bAssetIndexes, _bAsset, _flag);
    }
```

Listing 3.3: Multiple Setters in CurvedMasset

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, we notice that there are at least two parameters do not have the corresponding `setters`: `forgeValidatorLocked` and `basket.failed`. In other words, they remain constant during the entire protocol operation.

Recommendation Revisit those `setters` for current risk parameters and ensure they are adjustable. Otherwise, no need to keep them as as part of risk parameters.

Status This issue has been confirmed. Consider the related states are not doing any harm and merely informational, the team decides to leave as is, in case they may be used in the future for their original purpose.

3.4 Same Function Name With Different Definitions

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CurvedMasset`, `Manager`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

As mentioned in Section 3.2, the `Manager` contract is designed to perform the `Basket` management duties for an `mAsset` and also abstract certain logic to avoid bytecode inflation that prevents smooth deployment.

During our analysis of the `Manager` contract, we notice there is a helper `_getAsset()` that shares a similar functionality of a routine with the same function name in the `CurvedMasset` contract. To elaborate, we show below the code snippet of `_getAsset()` in the two contracts. As the name indicates, this routine is responsible for obtaining the current `bAsset` in terms of its index in the internal records of `bAssetPersonal` and `bAssetData`.

```
    /**
     * @dev Gets a bAsset from storage
     * @param _asset         Address of the asset
     * @return idx          Index of the asset
     * @return personal     Personal details for the asset
     */
    function _getAsset(address _asset) internal view returns (uint8 idx, BassetPersonal
        memory personal) {
        idx = bAssetIndexes[_asset];
        personal = bAssetPersonal[idx];
        require(personal.addr == _asset, "Invalid asset input");
    }
```

Listing 3.4: CurvedMasset::_getAsset()

```
    /**
     * @dev Gets a bAsset from storage
     * @param _asset       Address of the asset
     * @return idx          Index of the asset
     */
    function _getAsset(
        MassetStructs.BassetPersonal[] storage _bAssetPersonal,
        mapping(address => uint8) storage _bAssetIndexes,
        address _asset
    ) internal view returns (uint8 idx) {
        idx = _bAssetIndexes[_asset];
        require(_bAssetPersonal[idx].addr == _asset, "Invalid asset input");
```

```
    }
```

Listing 3.5: Manager::_getAsset()

Apparently, though this routine shares the same name, it has different definition and implementation. With that, we suggest to rename a routine to another in order to avoid unnecessary confusion and facilitate future maintenance.

Recommendation Avoid using the same function name if there is a different definition/implementation.

Status The issue has been fixed by this commit: `754e7db`.

3.5 Removal Of Unused Code/Events

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CurvedMasset`, `Manager`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

mStable-ICSMM makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `StableMath`, and `SafeCast`, to facilitate its code implementation and organization. For example, the `CurvedMasset` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `CurvedMasset` contract (see the code snippet below), there are a function named `_noDuplicates()` that is defined, but not used.

```
    /**
     * @dev Requires that a given index array does not contain any duplicate keys
     * @param _count     Pre-calculated length of array
     * @param _assets    Array of bAsset addresses
     */
    function _noDuplicates(uint256 _count, address[] memory _assets) internal pure {
        for (uint256 i = 0; i < _count; i++) {
            for (uint256 j = i + 1; j < _count; j++) {
                require(_assets[i] != _assets[j], "No duplicate assets allowed");
            }
        }
    }
}
```

Listing 3.6: CurvedMasset::_noDuplicates()

In the meantime, the current implementation also defines two events that are not used: the `RedemptionFeeChanged()` event in `CurvedMasset` and the `BasketStatusChanged()` event in `Manager`.

Recommendation Remove the above-mentioned function and events that are not used.

Status The issue has been fixed by this commit: `754e7db`.

3.6 Improved Corner Cases in Root()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Root`
- Category: Coding Practices [4]
- CWE subcategory: CWE-561 [2]

Description

The mStable-ICSMM protocol has developed a new `ICSMM` design where there are upper and lower soft and hard limits for `bAsset` weights. Accordingly, the invariant that needs to be maintained has been adjusted with related penalty/bonus functions. The invariant computation often needs to calculate the integer square root of a given number, i.e., the familiar `sqrt()` function. The `sqrt()` function, implemented in `Root`, follows the `Babylonian` method for calculating the integer square root. Specifically, for a given x, we need to find out the largest integer z such that $z^2 <= x$.

```
5    /**
6     * @dev Returns the square root of a given number
7     * @param x Input
8     * @return y Square root of Input
9     */
10   function sqrt(uint256 x) internal pure returns (uint256 y) {
11       uint256 z = (x + 1) / 2;
12       y = x;
13       while (z < y) {
14           y = z;
15           z = ((x / z) + z) / 2;
16       }
17   }
```

Listing 3.7: Root::sqrt()

We show above current `sqrt()` implementation. The initial value of z to the iteration was given as $z = (x + 1)/2$, which results in an integer overflow when $x = uint256(-1)$. In other words, the overflow essentially sets z to zero, leading to a `division by zero` in the calculation of $z = (x/z + z)/2$ (line 25).

Note that this does not result in an incorrect return value from `sqrt()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a `divide by zero`, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss.

To address this particular corner case, We suggest to change the initial value to $z = x/2 + 1$, making `sqrt()` well defined over its all possible inputs. Also, if necessary, there is an optimized implementation from `Uniswap-lib` with the following commit hash: 99f3f28.

Recommendation Revise the above calculation to avoid the unnecessary integer overflow.

Status The issue has been fixed by this commit: `754e7db`.

3.7 Potential State Inconsistency From Memory Cache

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `CurvedMasset`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1099 [1]

Description

In mStable-ICSMM, the `CurvedMasset` contract is the main entry point for liquidity providers to participate in the pool and the returned pool tokens can be staked in the `savings contract` for accumulated system yields. In this section, we focus on the deposit/withdraw logic for liquidity providers.

To elaborate, we show below the `_mintTo()` routine. This routine implements a rather straightforward logic in firstly validating the input and collecting the funds from the user, then transferring to `platform integrator` (if configured), next calculating the minted amount (in `mAssetMinted` - line 332), and finally minting the amount in the credit of the liquidity provider to represent the ownership on the pool.

```
305     /** @dev Mint Single */
306     function _mintTo(
307         address _input,
308         uint256 _inputQuantity,
309         uint256 _minMassetQuantity,
310         address _recipient
311     ) internal returns (uint256 mAssetMinted) {
312         require(_recipient != address(0), "Must be a valid recipient");
313         require(_inputQuantity > 0, "Quantity must not be zero");
```

```

        BassetData[] memory allBassets = bAssetData;
        (uint8 bAssetIndex, BassetPersonal memory personal) = _getAsset(_input);

        Cache memory cache = _getCacheDetails();

        // Transfer collateral to the platform integration address and call deposit
        uint256 quantityDeposited =
            _depositTokens(
                personal.addr,
                allBassets[bAssetIndex].ratio,
                personal.integrator,
                personal.hasTxFee,
                _inputQuantity,
                cache.maxCache
            );

        // Validation should be after token transfer, as bAssetQty is unknown before
        mAssetMinted = forgeValidator.computeMint(allBassets, bAssetIndex,
            quantityDeposited);
        require(mAssetMinted >= _minMassetQuantity, "Mint mAsset quantity < min qty");

        // Log the Vault increase - can only be done when basket is healthy
        bAssetData[bAssetIndex].vaultBalance =
            allBassets[bAssetIndex].vaultBalance +
            SafeCast.toUint128(quantityDeposited);

        // Mint the Masset
        _mint(_recipient, mAssetMinted);
        emit Minted(msg.sender, _recipient, mAssetMinted, _input, quantityDeposited);
    }
```

Listing 3.8: CurvedMasset::_mintTo()

It comes to our attention that the global state of `bAssetData` is preloaded into memory and cached in `allBassets`, presumably for the benefit of avoiding the repeated readings from the storage (and the associated gas cost). The cached memory copy is further used to compute the resulting balance of the pool. In between, the handling logic may interact with the user from the deposited funds and the external `platform integrator` for potential system yields. These external interactions inevitably expose the risk of being reentered to possible manipulation of balance.

In the meantime, we highlight that the current implementation has properly been enforced with necessary reentrancy protection. And there is not a functional problem so far. However, the recent incident of the `Cover` exploit [8] was due to a vulnerability with the same nature: a cached memory state is used to calculate the amount for reward. In other words, if the cached memory state becomes stale (due to possible reentrancy or other means), it is a critical vulnerability that will likely occur. With that, instead of using the cached state for the final balance calculation, it is suggested to read from the storage for the latest balance, effectively ruling out the potential risk.

Recommendation Ensure the freshness of the underlying asset balance for the computation and storage of the updated balance.

Status The issue has been fixed by this commit: `754e7db`.

3.8 Mixed Risk Parameters Between swapFee And redemptionFee

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CurvedMasset`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.3, DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand and the mStable-ICSMM protocol is no exception. In this section, we examine two specific system-wide risk parameters: `swapFee`, and `redemptionFee`.

As their names indicate, the `swapFee` parameter determines the fee that will be charged for traders that intend to swap from a token to another while the `redemptionFee` parameter indicates the fee that will be charged for liquidity providers when they intend to withdraw previously deposited liquidity.

To elaborate, we show below the `_redeemMasset()` routine that handles the request from liquidity providers to withdraw their liquidity. It comes to our attention that the fee is charged based on the `swapFee` parameter, not `redemptionFee`.

```
    /**
     * @dev Redeem mAsset for a multiple bAssets
     */
    function _redeemMasset(
        address _output,
        uint256 _inputQuantity,
        uint256 _minOutputQuantity,
        address _recipient
    ) internal returns (uint256 bAssetQuantity) {
        require(_inputQuantity > 0, "Invalid redemption quantity");

        // Load the bAsset data from storage into memory
        BassetData[] memory allBassets = bAssetData;
        (uint8 bAssetIndex, BassetPersonal memory personal) = _getAsset(_output);

        Cache memory cache = _getCacheDetails();

        // Calculate redemption quantities
```

```
        uint256 scaledFee = _inputQuantity.mulTruncate(swapFee);
        bAssetQuantity = forgeValidator.computeRedeem(
            allBassets,
            bAssetIndex,
            _inputQuantity - scaledFee
        );
        require(bAssetQuantity >= _minOutputQuantity, "bAsset qty < min bAsset qty");

        // Apply fees, burn mAsset and return bAsset to recipient
        // 1.0. Burn the full amount of Masset
        _burn(msg.sender, _inputQuantity);
        // 2.0. Transfer the Bassets to the recipient and count fees
        _withdrawTokens(
            bAssetQuantity,
            personal,
            allBassets[bAssetIndex],
            _recipient,
            cache.maxCache
        );
        // 3.0. Set vault balance and surplus
        bAssetData[bAssetIndex].vaultBalance =
            allBassets[bAssetIndex].vaultBalance -
            SafeCast.toUint128(bAssetQuantity);
        surplus = cache.surplus + scaledFee;

        emit Redeemed(
            msg.sender,
            _recipient,
            _inputQuantity,
            personal.addr,
            bAssetQuantity,
            scaledFee
        );
    }
```

Listing 3.9: CurvedMasset::_redeemMasset()

Recommendation Use the `redemptionFee` parameter for the calculation of the associated redemption cost, unless it is designed that `redemptionFee` should always be identical with `swapFee`. The presence of both parameters indicates the separation of `redemptionFee` from `swapFee` is intended and meaningful.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `mStable-ICSMM` protocol. The audited system presents a unique addition to current DeFi offerings by proposing a new `Incentivized Constant Sum Market Maker` (or `ICSMM`) design. The current code base is well organized and neatly engineered. Those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[8] PeckShield. Cover Incident: The Unlimited Token-Minting Vulnerability. https://peckshield.medium.com/cover-incident-the-unlimited-token-minting-vulnerability-f3afd9d2405c.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.