

Alpaca Finance 2.0 Money Market

Smart Contract Audit Report
Prepared for Alpaca Finance



Date Issued:	Mar 28, 2023
Project ID:	AUDIT2023003
Version:	v1.0
Confidentiality Level:	Public

Report Information

Project ID	AUDIT2023003
Version	v1.0
Client	Alpaca Finance
Project	Alpaca Finance 2.0 Money Market
Auditor(s)	Patipon Suwanbol Puttimet Thammasaeng Phitchakorn Apiratisakul Fungkiat Phadejtaku
Author(s)	Patipon Suwanbol Puttimet Thammasaeng Phitchakorn Apiratisakul Fungkiat Phadejtaku
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Mar 28, 2023	Full report	Patipon Suwanbol Puttimet Thammasaeng Phitchakorn Apiratisakul Fungkiat Phadejtaku

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary	1
1.1. Audit Result	1
1.2. Disclaimer	1
2. Project Overview	2
2.1. Project Introduction	2
2.2. Scope	3
3. Methodology	7
3.1. Test Categories	7
3.2. Audit Items	8
3.3. Risk Rating	10
4. Summary of Findings	11
5. Detailed Findings Information	13
5.1. Centralized Control of State Variable	13
5.2. Use of Upgradable Contract Design	17
5.3. Denial of Service in repay() Function	19
5.4. Missing Input Validation in setPoolRewarders	28
5.5. Denial of Service in the accrueNonCollatInterest() Function	31
5.6. Lack of Bad Debt Token Recovery after Write-Off	34
5.7. Unable to Fully Repurchase Debt Share on Borrow Position	41
5.8. Design Flaw in Supporting Deflationary Token	46
5.9. Inconsistent Interest Accrual Frequency	54
5.10. Incorrect Logging Parameter	57
5.11. Inconsistent Usage of Function Caller Identifiers	64
5.12. Outdated Compiler Version	66
6. Appendix	69
6.1. About Inspex	69

1. Executive Summary

As requested by Alpaca Finance, Inspex team conducted an audit to verify the security posture of the Alpaca Finance 2.0 Money Market smart contracts between Mar 1, 2023 and Mar 13, 2023. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Alpaca Finance 2.0 Money Market smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 high, 5 medium, 2 low, and 3 info-severity issues. With the project team's prompt response, 2 high, 5 medium, 1 low and 2 info-severity issues were resolved or mitigated in the reassessment, while 1 low and 1 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that Alpaca Finance 2.0 Money Market smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.



1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Alpaca Finance 2.0 (Money Market) offers a comprehensive suite of DeFi features, including lending and borrowing services. While providing liquidity to the platform, the user will receive the interest bearing token which can be further used as collateral for borrowing other underlying tokens. This introduces a new strategy for farming. Additionally, while borrowing tokens, users can earn extra yield in the form of governance tokens and other token rewards, maximizing the yield.

Scope Information:

Project Name	Alpaca Finance 2.0 Money Market
Website	https://www.alpacafinance.org/
Smart Contract Type	Ethereum Smart Contract
Chain	BNB Smart Chain
Programming Language	Solidity
Category	Yield Farming, Auto Compound, Token, Lending

Audit Information:

Audit Method	Whitebox
Audit Date	Mar 1, 2023 - Mar 13, 2023
Reassessment Date	Mar 24, 2023 - Mar 27, 2023

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox:** The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox:** Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 5bdfc8c8ce1026b290fdeb4179f16a0754424bd1)

Contract	Location (URL)
DebtToken	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/DebtToken.sol
InterestBearingToken	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/InterestBearingToken.sol
MoneyMarketDiamond	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/MoneyMarketDiamond.sol
PancakeswapV2Liqui dationStrategy	<a href="https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/PancakeswapV2Liqui
dationStrategy.sol">https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/PancakeswapV2Liqui dationStrategy.sol
PancakeswapV2Liqui dationStrategy	<a href="https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/PancakeswapV2Liqui
dationStrategy.sol">https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/PancakeswapV2Liqui dationStrategy.sol
AdminFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/AdminFacet.sol
BorrowFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/BorrowFacet.sol
CollateralFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/CollateralFacet.sol
DiamondCutFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/DiamondCutFacet.sol
DiamondLoupeFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/DiamondLoupeFacet.sol
LendFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/LendFacet.sol
LiquidationFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/LiquidationFacet.sol
NonCollatBorrowFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/NonCollatBorrowFacet.sol
OwnershipFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/OwnershipFacet.sol

ViewFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/facets/ViewFacet.sol
FixedFeeModel	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/fee-models/FixedFeeModel.sol
FixedInterestRateModel	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/interest-models/FixedInterestRateModel.sol
TripleSlopeModel6	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/interest-models/TripleSlopeModel6.sol
TripleSlopeModel7	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/interest-models/TripleSlopeModel7.sol
LibDiamond	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/libraries/LibDiamond.sol
LibDoublyLinkedList	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/libraries/LibDoublyLinkedList.sol
LibFullMath	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/libraries/LibFullMath.sol
LibMoneyMarket01	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/libraries/LibMoneyMarket01.sol
LibReentrancyGuard	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/libraries/LibReentrancyGuard.sol
LibSafeToken	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/libraries/LibSafeToken.sol
LibShareUtil	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/libraries/LibShareUtil.sol
AlpacaV2Oracle	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/oracle/AlpacaV2Oracle.sol
MoneyMarketAccountManager	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/account-manager/MoneyMarketAccountManager.sol
MiniFL	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/miniFL/MiniFL.sol
Rewarder	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/miniFL/Rewarder.sol

Reassessment: (Commit: 48a6bd6491ff730ba5b596cdc073f01fc8bcc53c)

Contract	Location (URL)
DebtToken	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/DebtToken.sol
InterestBearingToken	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/InterestBearingToken.sol
MoneyMarketDiamond	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/MoneyMarketDiamond.sol
PancakeswapV2IbTokenLiquidationStrategy	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/PancakeswapV2IbTokenLiquidationStrategy.sol
PancakeswapV2LiquidationStrategy	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/PancakeswapV2LiquidationStrategy.sol
AdminFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/AdminFacet.sol
BorrowFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/BorrowFacet.sol
CollateralFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/CollateralFacet.sol
DiamondCutFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/MMDiamondCutFacet.sol
DiamondLoupeFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/MMDiamondLoupeFacet.sol
LendFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/LendFacet.sol
LiquidationFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/LiquidationFacet.sol
NonCollatBorrowFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/NonCollatBorrowFacet.sol
OwnershipFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/MMOwnershipFacet.sol
ViewFacet	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/facets/ViewFacet.sol

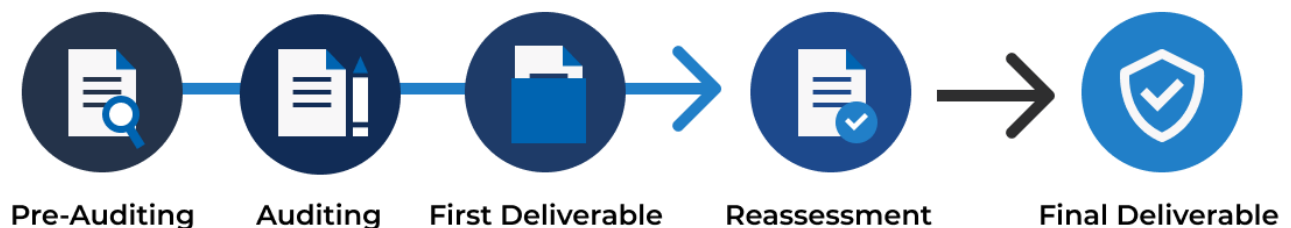
FixedFeeModel	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/fee-models/FixedFeeModel.sol
FixedInterestRateModel	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/interest-models/FixedInterestRateModel.sol
TripleSlopeModel6	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/interest-models/TripleSlopeModel6.sol
TripleSlopeModel7	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/interest-models/TripleSlopeModel7.sol
LibDiamond	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/libraries/LibDiamond.sol
LibDoublyLinkedList	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/libraries/LibDoublyLinkedList.sol
LibFullMath	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/libraries/LibFullMath.sol
LibMoneyMarket01	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/libraries/LibMoneyMarket01.sol
LibReentrancyGuard	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/libraries/LibReentrancyGuard.sol
LibSafeToken	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/libraries/LibSafeToken.sol
LibShareUtil	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/libraries/LibShareUtil.sol
AlpacaV2Oracle	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/oracle/AlpacaV2Oracle.sol
MoneyMarketAccountManager	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/account-manager/MoneyMarketAccountManager.sol
MiniFL	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/miniFL/MiniFL.sol
Rewarder	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/miniFL/Rewarder.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing:** Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing:** Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting:** Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment:** Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable:** Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at <https://inspex.gitbook.io/testing-guide/>.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	<ul style="list-style-type: none">6.1. Unknown external components should not be invoked6.2. Funds should not be approved or transferred to unknown accounts6.3. Reentrant calling should not negatively affect the contract states6.4. Vulnerable or outdated components should not be used in the smart contract6.5. Deprecated components that have no longer been supported should not be used in the smart contract6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	<ul style="list-style-type: none">7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows7.2. Explicit conversion of types should be checked to prevent unexpected results7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	<ul style="list-style-type: none">8.1. State changing functions that loop over unbounded data structures should not be used8.2. Unexpected revert should not make the whole smart contract unusable8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	<ul style="list-style-type: none">9.1. State and function visibility should be explicitly labeled9.2. Token implementation should comply with the standard specification9.3. Floating pragma version should not be used9.4. Builtin symbols should not be shadowed9.5. Functions that are never called internally should not have public visibility9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood:** a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact:** a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

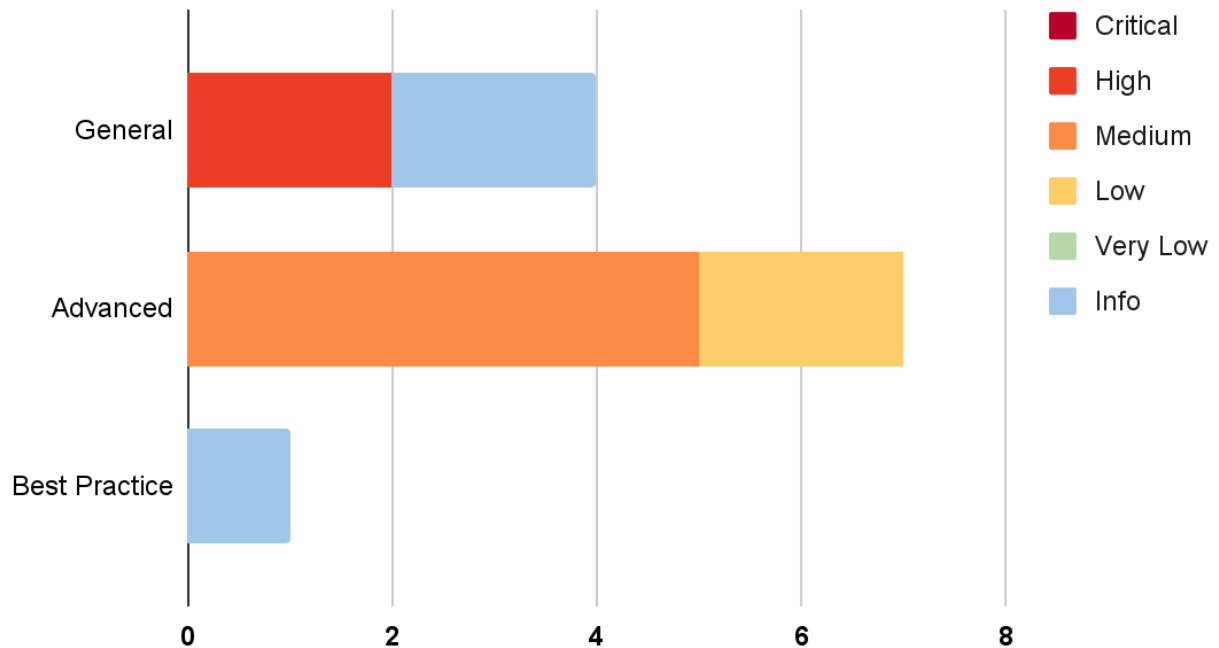
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Likelihood Impact	Likelihood		
	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

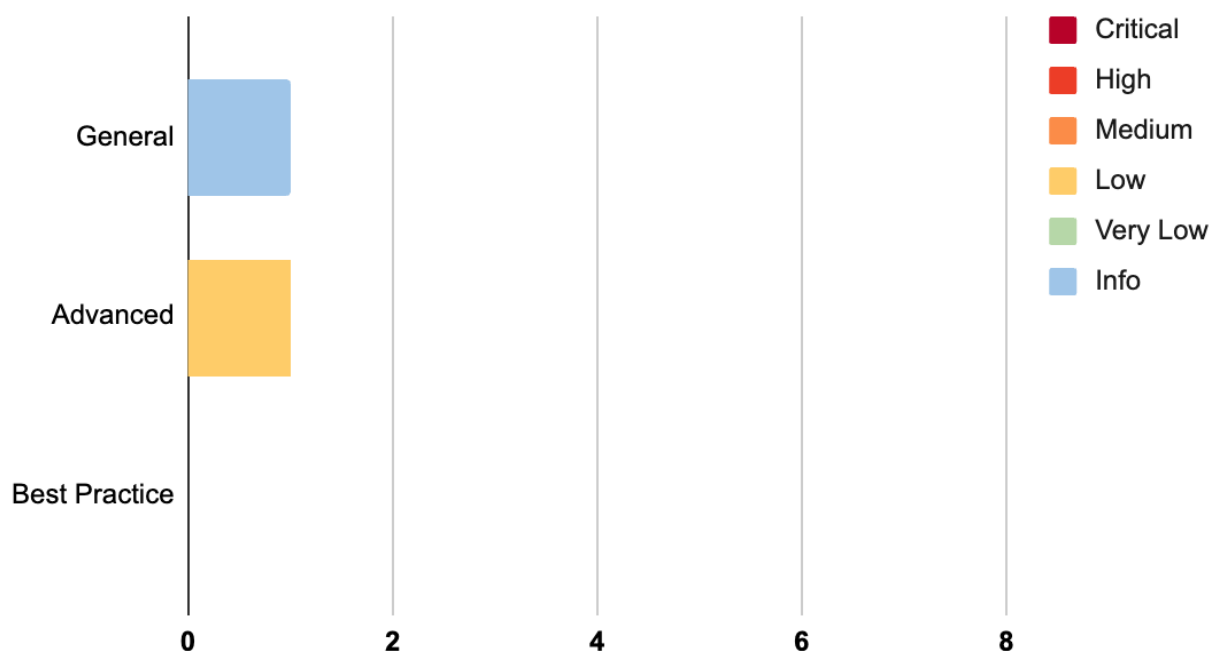
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Centralized Control of State Variable	General	High	Resolved *
IDX-002	Use of Upgradable Contract Design	General	High	Resolved *
IDX-003	Denial of Service in repay() Function	Advanced	Medium	Resolved
IDX-004	Missing Input Validation in setPoolRewarders	Advanced	Medium	Resolved
IDX-005	Denial of Service in the accrueNonCollatInterest() Function	Advanced	Medium	Resolved
IDX-006	Lack of Bad Debt Token Recovery after Write-Off	Advanced	Medium	Resolved
IDX-007	Unable to Fully Repurchase Debt Share on Borrow Position	Advanced	Medium	Resolved
IDX-008	Design Flaw in Supporting Deflationary Token	Advanced	Low	Resolved
IDX-009	Inconsistent Interest Accrual Frequency	Advanced	Low	Acknowledged
IDX-010	Incorrect Logging Parameter	General	Info	No Security Impact
IDX-011	Inconsistent Usage of Function Caller Identifiers	Best Practice	Info	Resolved
IDX-012	Outdated Compiler Version	General	Info	Resolved

* The mitigations or clarifications by Alpaca Finance can be found in Chapter 5.

5. Detailed Findings Information

5.1. Centralized Control of State Variable

ID	IDX-001
Target	MiniFL Rewarder AdminFacet DiamondCutFacet OwnershipFacet PancakeswapV2IbTokenLiquidationStrategy PancakeswapV2LiquidationStrategy
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	Severity: High Impact: High The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. Likelihood: Medium There is nothing to restrict the changes from being done; however, this action can only be done by the privileged roles.
Status	Resolved * The Alpaca Finance team has mitigated this issue by implementing a Timelock contract as the owner of all contracts to prevent immediate changes or upgrades to the contract and also to provide transparency in the process of maintaining contract upgrades. However, the timelock mechanism was not in use at the time of the reassessment. Therefore, Inspex suggests the platform users to confirm the usage of the timelock mechanism before using the platform.

5.1.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
MiniFL.sol (L:131)	MiniFL	setPool()	onlyOwner
MiniFL.sol (L:149)	MiniFL	setAlpacaPerSecond()	onlyOwner
MiniFL.sol (L:400)	MiniFL	setMaxAlpacaPerSecond()	onlyOwner
MiniFL.sol (L:411)	MiniFL	setPoolRewarders()	onlyOwner
AdminFacet.sol (L:83)	AdminFacet	openMarket()	onlyOwner
AdminFacet.sol (L:145)	AdminFacet	setTokenConfigs()	onlyOwner
AdminFacet.sol (L:206)	AdminFacet	setNonCollatBorrowerOk()	onlyOwner
AdminFacet.sol (L:215)	AdminFacet	setInterestModel()	onlyOwner
AdminFacet.sol (L:228)	AdminFacet	setNonCollatInterestModel()	onlyOwner
AdminFacet.sol (L:246)	AdminFacet	setOracle()	onlyOwner
AdminFacet.sol (L:257)	AdminFacet	setRepurchasersOk()	onlyOwner
AdminFacet.sol (L:272)	AdminFacet	setLiquidationStratsOk()	onlyOwner
AdminFacet.sol (L:287)	AdminFacet	setLiquidatorsOk()	onlyOwner
AdminFacet.sol (L:302)	AdminFacet	setAccountManagersOk()	onlyOwner
AdminFacet.sol (L:316)	AdminFacet	setLiquidationTreasury()	onlyOwner
AdminFacet.sol (L:329)	AdminFacet	withdrawProtocolReserve()	onlyOwner
AdminFacet.sol (L:358)	AdminFacet	setFees()	onlyOwner
AdminFacet.sol (L:387)	AdminFacet	setRepurchaseRewardModel()	onlyOwner
AdminFacet.sol (L:401)	AdminFacet	setLbTokenImplementation()	onlyOwner
AdminFacet.sol (L:412)	AdminFacet	setDebtTokenImplementation()	onlyOwner

AdminFacet.sol (L:423)	AdminFacet	setProtocolConfigs()	onlyOwner
AdminFacet.sol (L:463)	AdminFacet	setLiquidationParams()	onlyOwner
AdminFacet.sol (L:481)	AdminFacet	setMaxNumOfToken()	onlyOwner
AdminFacet.sol (L:496)	AdminFacet	setMinDebtSize()	onlyOwner
AdminFacet.sol (L:505)	AdminFacet	writeOffSubAccountsDebt()	onlyOwner
AdminFacet.sol (L:572)	AdminFacet	setEmergencyPaused()	onlyOwner
DiamondCutFacet.sol (L:22)	DiamondCutFacet	diamondCut()	diamondStorage().contractOwner
OwnershipFacet.sol (L:14)	OwnershipFacet	transferOwnership()	diamondStorage().contractOwner
OwnershipFacet.sol (L:25)	OwnershipFacet	acceptOwnership()	LibDiamond.pendingOwner()
PancakeswapV2IbTokenLiquidationStrategy.sol (L:135)	PancakeswapV2IbTokenLiquidationStrategy	setPaths()	onlyOwner
PancakeswapV2IbTokenLiquidationStrategy.sol (L:155)	PancakeswapV2IbTokenLiquidationStrategy	setCallersOk()	onlyOwner
PancakeswapV2LiquidationStrategy.sol (L:85)	PancakeswapV2LiquidationStrategy	setPaths()	onlyOwner
PancakeswapV2LiquidationStrategy.sol (L:105)	PancakeswapV2LiquidationStrategy	setCallersOk()	onlyOwner
Rewardder.sol (L:191)	Rewardder	setRewardPerSecond()	onlyOwner
Rewardder.sol (L:203)	Rewardder	addPool()	onlyOwner
Rewardder.sol (L:228)	Rewardder	setPool()	onlyOwner
Rewardder.sol (L:338)	Rewardder	setMaxRewardPerSecond()	onlyOwner

5.1.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests implementing a community-run smart contract governance to control the use of these functions.

If removing the functions or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time at least 24 hours.

5.2. Use of Upgradable Contract Design

ID	IDX-002
Target	AdminFacet BorrowFacet CollateralFacet DiamondCutFacet DiamondLoupeFacet LendFacet LiquidationFacet NonCollatBorrowFacet OwnershipFacet ViewFacet DebtToken InterestBearingToken MoneyMarketDiamond
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	Severity: High Impact: High The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the users' funds anytime. Likelihood: Medium The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the users' funds anytime.
Status	Resolved * The Alpaca Finance team has mitigated this issue by implementing a Timelock contract as the owner of all contracts to prevent immediate changes or upgrades to the contract and also to provide transparency in the process of maintaining contract upgrades. However, the timelock mechanism was not in use at the time of the reassessment. Therefore, Inspex suggests the platform users to confirm the usage of the timelock mechanism before using the platform.

5.2.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As these smart contracts are upgradable, the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy.

5.2.2. Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make the smart contracts upgradeable.

However, if upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes at least 24 hours on the proxy owner role. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.3. Denial of Service in repay() Function

ID	IDX-003
Target	BorrowFacet
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Medium</p> <p>Impact: Medium When the borrower wants to repay their borrowed amount with the debt token at that time, the transaction can be reverted, resulting in further interest payments for the borrower and increasing the <code>usedBorrowedPower</code> of the account.</p> <p>Likelihood: Medium When the user decides to repay the borrowed position with full debt amount and the platform has set the <code>minDebtSize</code> state to a value greater than zero. Also, the ratio for <code>moneyMarketDs.overCollatDebtValues</code> and <code>moneyMarketDs.overCollatDebtShares</code> must be indivisible.</p>
Status	<p>Resolved</p> <p>The Alpaca Finance team has resolved this issue by implementing the <code>shareToValueRoundingUp()</code> function and applying it to the <code>repay()</code> function in commit <code>48a6bd6491ff730ba5b596cdc073f01fc8bcc53c</code>.</p>

5.3.1. Description

The `repayFor()` function in the `MoneyMarketAccountManager` contract is utilized to repay a user's debt. To repay the debt, the user specifies the `_debtShareToRepay` debt share and the `_repayAmount` amount they want to repay. Then, the function calls the `repay()` function in the `BorrowFacet` contract on line 343.

MoneyMarketAccountManager.sol

```

325 function repayFor(
326     address _account,
327     uint256 _subAccountId,
328     address _token,
329     uint256 _repayAmount,
330     uint256 _debtShareToRepay
331 ) external {
332     // cache the balance of token before proceeding
333     uint256 _amountBefore = IERC20(_token).balanceOf(address(this));
334
335     // Fund this contract from caller
336     // ignore the fact that there might be fee on transfer

```

```

337     IERC20(_token).safeTransferFrom(msg.sender, address(this), _repayAmount);
338
339     // Call repay by forwarding input _debtShareToRepay
340     // Money Market should deduct the fund as much as possible
341     // If there's excess amount left, transfer back to user
342     IERC20(_token).safeApprove(address(moneyMarket), _repayAmount);
343     moneyMarket.repay(_account, _subAccountId, _token, _debtShareToRepay);
344     // Reset allowance as moneyMarket.repay() might not use all the allowance
345     IERC20(_token).safeApprove(address(moneyMarket), 0);
346
347     // Calculate the excess amount left in the contract
348     // This will revert if the input repay amount has lower value than
    _debtShareToRepay
349     // And there's some token left in contract (can be done by inject token
    directly to this contract)
350     uint256 _excessAmount = IERC20(_token).balanceOf(address(this)) -
    _amountBefore;
351
352     if (_excessAmount != 0) {
353         IERC20(_token).safeTransfer(msg.sender, _excessAmount);
354     }
355 }

```

In the `repay()` function, the debt share state `_debtShareToRepay` is used to calculate the `_actualAmountToRepay` token amount required to repay the specific share (line 129). The contract then pulls the tokens that the user has transferred to the `MoneyMarketAccountManager` before (line 134), and uses the `valueToShare()` function to calculate the `_actualShareToRepay`, which is converted from the `_actualAmountToRepay` amount to a share (line 137).

BorrowFacet.sol

```

93 function repay(
94     address _account,
95     uint256 _subAccountId,
96     address _token,
97     uint256 _debtShareToRepay
98 ) external nonReentrant {
99     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
    LibMoneyMarket01.moneyMarketDiamondStorage();
100
101     // This function should not be called from anyone
102     // except account manager contract and will revert upon trying to do so
103     LibMoneyMarket01.onlyAccountManager(moneyMarketDs);
104
105     address _subAccount = LibMoneyMarket01.getSubAccount(_account,
    _subAccountId);
106

```

```
107 // accrue all debt tokens under subaccount
108 // because used borrowing power is calculated from all debt token of sub
account
109 LibMoneyMarket01.accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);
110
111 // Get the current debt amount and share of this token under the subaccount
112 // The current debt share will be used to cap the maximum that can be repaid
113 // The current debt amount will be used to check the minimum debt size after
repaid
114 (uint256 _currentDebtShare, uint256 _currentDebtAmount) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
115     _subAccount,
116     _token,
117     moneyMarketDs
118 );
119
120 // The debt share that can be repaid should not exceed the current debt share
121 // that the subaccount is holding
122 uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare,
_debtShareToRepay);
123
124 // caching these variables to save gas from multiple reads
125 uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
126 uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];
127
128 // Find the actual underlying amount that need to be pulled from the share
129 uint256 _actualAmountToRepay = LibShareUtil.shareToValue(_actualShareToRepay,
_cachedDebtValue, _cachedDebtShare);
130
131 // Pull the token from the account manager, the actual amount received will
be used for debt accounting
132 // In case somehow there's fee on transfer - which's might be introduced
after the token was lent
133 // Not reverting to ensure that repay transaction can be done even if there's
fee on transfer
134 _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender,
_actualAmountToRepay);
135
136 // Recalculate the debt share to remove in case there's fee on transfer
137 _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay,
_cachedDebtShare, _cachedDebtValue);
138
139 // Increase the reserve amount of the token as there's new physical token
coming in
140 moneyMarketDs.reserves[_token] += _actualAmountToRepay;
141
142 // Check and revert if the repay transaction will violate the business rule
```



```

143 // namely the debt size after repaid should be more than minimum debt size
144 _validateRepay(
145     _token,
146     _currentDebtShare,
147     _currentDebtAmount,
148     _actualShareToRepay,
149     _actualAmountToRepay,
150     moneyMarketDs
151 );
152
153 // Remove the debt share from this subaccount's accounting
154 // additionally, this library call will unstake the debt token
155 // from miniFL and burn the debt token
156 LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
157     _account,
158     _subAccount,
159     _token,
160     _actualShareToRepay,
161     _actualAmountToRepay,
162     moneyMarketDs
163 );
164
165 emit LogRepay(_account, _subAccountId, _token, msg.sender,
166     _actualAmountToRepay);
167 }

```

The issue occurs when the `repay()` function calculates the amount required to repay the debt share using `shareToValue()` and then applies this value to calculate the share value again through the `valueToShare()` function. Hence, if there is a precision loss from the first calculation, the value after this will be rounded off. Furthermore, the debt share calculation from the first time that was calculated from the `overCollatBorrow()` function has already been rounded up as in lines 1033 - 1037, increasing the chance that the result from converting share back to value (`shareToValue()`) back will be rounded off.

LibMoneyMarket01.sol

```

1020 function overCollatBorrow(
1021     address _account,
1022     address _subAccount,
1023     address _token,
1024     uint256 _amount,
1025     MoneyMarketDiamondStorage storage moneyMarketDs
1026 ) internal returns (uint256 _shareToAdd) {
1027     LibDoublyLinkedList.List storage userDebtShare =
1028     moneyMarketDs.subAccountDebtShares[_subAccount];
1029     IMiniFL _miniFL = moneyMarketDs.miniFL;
1030     userDebtShare.initIfNotExist();

```

```

1031
1032 // get share value to update states
1033 _shareToAdd = LibShareUtil.valueToShareRoundingUp(
1034     _amount,
1035     moneyMarketDs.overCollatDebtShares[_token],
1036     moneyMarketDs.overCollatDebtValues[_token]
1037 );
1038
1039 // update over collat debt
1040 moneyMarketDs.overCollatDebtShares[_token] += _shareToAdd;
1041 moneyMarketDs.overCollatDebtValues[_token] += _amount;
1042
1043 // update global debt
1044 moneyMarketDs.globalDebts[_token] += _amount;
1045
1046 // update user's debtshare
1047 userDebtShare.addOrUpdate(_token, userDebtShare.getAmount(_token) +
    _shareToAdd);
1048 // revert if number of debt tokens exceed limit per sub account
1049 if (userDebtShare.length() > moneyMarketDs.maxNumOfDebtPerSubAccount) {
1050     revert LibMoneyMarket01_NumberOfTokenExceedLimit();
1051 }
1052
1053 // mint debt token to money market and stake to miniFL
1054 address _debtToken = moneyMarketDs.tokenToDebtTokens[_token];
1055 uint256 _poolId = moneyMarketDs.miniFLPoolIds[_debtToken];
1056
1057 // pool for debt token always exist
1058 // since pool is created during AdminFacet.openMarket()
1059 IDebtToken(_debtToken).mint(address(this), _shareToAdd);
1060 IERC20(_debtToken).safeApprove(address(_miniFL), _shareToAdd);
1061 _miniFL.deposit(_account, _poolId, _shareToAdd);
1062
1063 emit LogOverCollatBorrow(msg.sender, _subAccount, _token, _amount,
    _shareToAdd);
1064 }

```

Hence, the calculation in the `_validateRepay()` function will result in the `revert BorrowFacet_BorrowLessThanMinDebtSize()`; line since the `_currentSubAccountDebtShare` is greater than the `_shareToRepay` due to the rounded off value from `_shareToRepay` in 1 wei. The `_currentSubAccountDebtAmount` value will be equal to the `_amountToRepay` since this is the full repayment, so it results in zero.

As a result, the equation will check whether `if (0 < moneyMarketDs.minDebtSize)` is true or not. So if the `moneyMarketDs.minDebtSize` is not zero, it will trigger the transaction to be reverted.

BorrowFacet.sol

```

254 function _validateRepay(
255     address _repayToken,
256     uint256 _currentSubAccountDebtShare,
257     uint256 _currentSubAccountDebtAmount,
258     uint256 _shareToRepay,
259     uint256 _amountToRepay,
260     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs
261 ) internal view {
262     // if partial repay, check if debt after repaid more than minDebtSize
263     // no check if repay entire debt
264     if (_currentSubAccountDebtShare > _shareToRepay) {
265         uint256 _tokenPrice = LibMoneyMarket01.getPriceUSD(_repayToken,
moneyMarketDs);
266         // Revert if debt size in USD after repaid is less than minDebtSize
267         if (
268             ((_currentSubAccountDebtAmount - _amountToRepay) *
269             moneyMarketDs.tokenConfigs[_repayToken].to18ConversionFactor *
270             _tokenPrice) /
271             1e18 <
272             moneyMarketDs.minDebtSize
273         ) {
274             revert BorrowFacet_BorrowLessThanMinDebtSize();
275         }
276     }
277 }

```

5.3.2. Remediation

Inspex suggests avoiding the precision loss when calculating the `_actualAmountToRepay` by adding the `shareToValueRoundingUp()` function instead (lines 18-29). This is used to cover the extra share that was added in the `overCollatBorrow()` function before when the debt share is calculated.

LibShareUtil.sol

```

1 // SPDX-License-Identifier: BUSL
2 pragma solidity 0.8.17;
3
4 import { LibFullMath } from "./LibFullMath.sol";
5
6 library LibShareUtil {
7     function shareToValue(
8         uint256 _shareAmount,
9         uint256 _totalValue,
10        uint256 _totalShare
11    ) internal pure returns (uint256) {
12        if (_totalShare == 0) {
13            return _shareAmount;

```

```
14     }
15     return LibFullMath.mulDiv(_shareAmount, _totalValue, _totalShare);
16 }
17
18 function shareToValueRoundingUp(
19     uint256 _shareAmount,
20     uint256 _totalValue,
21     uint256 _totalShare
22 ) internal pure returns (uint256) {
23     uint256 _values = shareToValue(_shareAmount, _totalValue, _totalShare);
24     uint256 _valueShares = valueToShare(_values, _totalShare, _totalValue);
25     if (_valueShares + 1 == _shareAmount) {
26         _values += 1;
27     }
28     return _values;
29 }
30
31 function valueToShare(
32     uint256 _tokenAmount,
33     uint256 _totalShare,
34     uint256 _totalValue
35 ) internal pure returns (uint256) {
36     if (_totalShare == 0) {
37         return _tokenAmount;
38     }
39     return LibFullMath.mulDiv(_tokenAmount, _totalShare, _totalValue);
40 }
41
42 function valueToShareRoundingUp(
43     uint256 _tokenAmount,
44     uint256 _totalShare,
45     uint256 _totalValue
46 ) internal pure returns (uint256) {
47     uint256 _shares = valueToShare(_tokenAmount, _totalShare, _totalValue);
48     uint256 _shareValues = shareToValue(_shares, _totalValue, _totalShare);
49     if (_shareValues + 1 == _tokenAmount) {
50         _shares += 1;
51     }
52     return _shares;
53 }
54 }
55 }
```

Apply it to the `repay()` function in the `BorrowFacet` contract.

BorrowFacet.sol

```
93 function repay(
```

```
94     address _account,
95     uint256 _subAccountId,
96     address _token,
97     uint256 _debtShareToRepay
98 ) external nonReentrant {
99     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
    LibMoneyMarket01.moneyMarketDiamondStorage();
100
101     // This function should not be called from anyone
102     // except account manager contract and will revert upon trying to do so
103     LibMoneyMarket01.onlyAccountManager(moneyMarketDs);
104
105     address _subAccount = LibMoneyMarket01.getSubAccount(_account,
    _subAccountId);
106
107     // accrue all debt tokens under subaccount
108     // because used borrowing power is calculated from all debt token of sub
    account
109     LibMoneyMarket01.accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);
110
111     // Get the current debt amount and share of this token under the subaccount
112     // The current debt share will be used to cap the maximum that can be repaid
113     // The current debt amount will be used to check the minimum debt size after
    repaid
114     (uint256 _currentDebtShare, uint256 _currentDebtAmount) =
    LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
115         _subAccount,
116         _token,
117         moneyMarketDs
118     );
119
120     // The debt share that can be repaid should not exceed the current debt share
121     // that the subaccount is holding
122     uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare,
    _debtShareToRepay);
123
124     // caching these variables to save gas from multiple reads
125     uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
126     uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];
127
128     // Find the actual underlying amount that need to be pulled from the share
129     uint256 _actualAmountToRepay =
    LibShareUtil.shareToValueRoundingUp(_actualShareToRepay, _cachedDebtValue,
    _cachedDebtShare);
130
131     // Pull the token from the account manager, the actual amount received will
    be used for debt accounting
```

```
132 // In case somehow there's fee on transfer - which's might be introduced
    after the token was lent
133 // Not reverting to ensure that repay transaction can be done even if there's
    fee on transfer
134 _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender,
    _actualAmountToRepay);
135
136 // Recalculate the debt share to remove in case there's fee on transfer
137 _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay,
    _cachedDebtShare, _cachedDebtValue);
138
139 // Increase the reserve amount of the token as there's new physical token
    coming in
140 moneyMarketDs.reserves[_token] += _actualAmountToRepay;
141
142 // Check and revert if the repay transaction will violate the business rule
143 // namely the debt size after repaid should be more than minimum debt size
144 _validateRepay(
145     _token,
146     _currentDebtShare,
147     _currentDebtAmount,
148     _actualShareToRepay,
149     _actualAmountToRepay,
150     moneyMarketDs
151 );
152
153 // Remove the debt share from this subaccount's accounting
154 // additionally, this library call will unstake the debt token
155 // from miniFL and burn the debt token
156 LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
157     _account,
158     _subAccount,
159     _token,
160     _actualShareToRepay,
161     _actualAmountToRepay,
162     moneyMarketDs
163 );
164
165 emit LogRepay(_account, _subAccountId, _token, msg.sender,
    _actualAmountToRepay);
166 }
```

5.4. Missing Input Validation in setPoolRewarders

ID	IDX-004
Target	MiniFL
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	<p>Severity: Medium</p> <p>Impact: High If there is a function that affects a borrowed position related to a token that provides extra yield through the Rewarder contract, any transactions related to that borrowed position will be reverted. This is because a function called <code>_updatePool()</code> will be triggered, which calls a non-existing pool in the Rewarder contract.</p> <p>Likelihood: Low It is unlikely that the platform will set pools in the Rewarder contract that do not match the same pool id on the MiniFL contract.</p>
Status	<p>Resolved</p> <p>The Alpaca Finance team has resolved this issue by adding input validation in the <code>setPoolRewarders()</code> function to ensure that the corresponding pool already exists in commit <code>48a6bd6491ff730ba5b596cdc073f01fc8bcc53c</code>.</p>

5.4.1. Description

The **MiniFL** contract is used to distribute rewards to borrowers and provides additional rewards through the **Rewarder** contract. The pool ID in the **MiniFL** contract must correspond to the **Rewarder** contract.

The platform owner is able to set the rewarder address via the `setPoolRewarders()` function, as shown below:

MiniFL.sol

```

411 function setPoolRewarders(uint256 _pid, address[] calldata _newRewarders)
    external onlyOwner {
412     uint256 _length = _newRewarders.length;
413     // loop to check rewarder should be belong to this MiniFL only
414     for (uint256 _i; _i < _length; ) {
415         if (IRewarder(_newRewarders[_i]).miniFL() != address(this)) {
416             revert MiniFL_BadRewarder();
417         }
418
419         unchecked {
420             ++_i;

```

```

421     }
422 }
423
424     rewarders[_pid] = _newRewarders;
425 }

```

However, the `setPoolRewarders()` does not validate that the pool is created in the `_newRewarders` contract.

As a result, the transaction related to the borrowed position will be reverted, if the `_poolInfo.lastRewardTime` is zero, due to the internal call to the `_updatePool()` function.

Rewarder.sol

```

274 function _updatePool(uint256 _pid) internal returns (PoolInfo memory) {
275     PoolInfo memory _poolInfo = poolInfo[_pid];
276
277     if (_poolInfo.lastRewardTime == 0) revert Rewarder1_PoolNotExisted();
278
279     if (block.timestamp > _poolInfo.lastRewardTime) {
280         uint256 _stakedBalance = IMiniFL(miniFL).getStakingReserves(_pid);
281         if (_stakedBalance > 0) {
282             uint256 _timePast;
283             unchecked {
284                 _timePast = block.timestamp - _poolInfo.lastRewardTime;
285             }
286             uint256 _rewards = (_timePast * rewardPerSecond * _poolInfo.allocPoint) /
totalAllocPoint;
287
288             // increase accRewardPerShare with `_rewards/stakedBalance` amount
289             // example:
290             // - oldaccRewardPerShare = 0
291             // - _rewards                = 2000
292             // - stakedBalance            = 10000
293             // _poolInfo.accRewardPerShare = oldaccRewardPerShare +
(_rewards/stakedBalance)
294             // _poolInfo.accRewardPerShare = 0 + 2000/10000 = 0.2
295             _poolInfo.accRewardPerShare =
296                 _poolInfo.accRewardPerShare +
297                 ((_rewards * ACC_REWARD_PRECISION) / _stakedBalance).toUint128();
298         }
299         _poolInfo.lastRewardTime = block.timestamp.toUint64();
300         poolInfo[_pid] = _poolInfo;
301         emit LogUpdatePool(_pid, _poolInfo.lastRewardTime, _stakedBalance,
_poolInfo.accRewardPerShare);
302     }
303     return _poolInfo;

```


304 }

5.4.2. Remediation

Inspex suggests adding the input validation in the `setPoolRewarders()` function to ensure that the corresponding pool exists in the `Rewarder` contract and that `_pid` is not zero.

MiniFL.sol

```
411 function setPoolRewarders(uint256 _pid, address[] calldata _newRewarders)
    external onlyOwner {
412     require(_pid != 0, "The pool id 0 is not allowed");
413     uint256 _length = _newRewarders.length;
414     // loop to check rewarder should be belong to this MiniFL only
415     for (uint256 _i; _i < _length; ) {
416         if (IRewarder(_newRewarders[_i]).miniFL() != address(this)) {
417             revert MiniFL_BadRewarder();
418         }
419         require(IRewarder(_newRewarders[_i]).poolInfo[_pid].lastRewardTime != 0,
            "The pool has not been created in the rewarder");
420
421
422         unchecked {
423             ++_i;
424         }
425     }
426
427     rewarders[_pid] = _newRewarders;
428 }
```

5.5. Denial of Service in the `accrueNonCollatInterest()` Function

ID	IDX-005
Target	BorrowFacet LendFacet CollateralFacet LiquidationFacet NonCollatBorrowFacet
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Medium</p> <p>Impact: High The <code>accrueNonCollatInterest()</code> function is called from all processes. If the number of non-collateral accounts is large and causes the gas limit to exceed, results in the denial of service.</p> <p>Likelihood: Low The issue is unlikely to occur, since it would require a large number of non-collateral accounts to cause the gas limit to exceed.</p>
Status	<p>Resolved</p> <p>The Alpaca Finance team has resolved this issue by implementing a cap for the amount of non-collateral accounts in commit <code>48a6bd6491ff730ba5b596cdc073f01fc8bcc53c</code>.</p>

5.5.1. Description

In the `LibMoneyMarket01` library, the `accrueNonCollatInterest()` function is used to accrue an interest from all non-collateral accounts.

The `accrueNonCollatInterest()` function is loop over the `moneyMarketDs.nonCollatTokenDebtValues[_token]` state which contain the list of all non-collateral account of the desire token as shown in lines 609 and 617-648.

LibMoneyMarket01.sol

```

603 function accrueNonCollatInterest(
604     address _token,
605     uint256 _timePast,
606     MoneyMarketDiamondStorage storage moneyMarketDs
607 ) internal returns (uint256 _totalNonCollatInterest) {
608     // get all non collat borrowers
609     LibDoublyLinkedList.Node[] memory _borrowedAccounts =
        moneyMarketDs.nonCollatTokenDebtValues[_token].getAll();

```

```
610     uint256 _accountLength = _borrowedAccounts.length;
611     address _account;
612     uint256 _currentAccountDebt;
613     uint256 _accountInterest;
614     uint256 _newNonCollatDebtValue;
615
616     // sum up all non collat interest of a token
617     for (uint256 _i; _i < _accountLength; ) {
618         _account = _borrowedAccounts[_i].token;
619         _currentAccountDebt = _borrowedAccounts[_i].amount;
620
621         // calculate interest
622         // calculation:
623         // _accountInterest = nonCollatInterestRate * _timePast *
624         // _currentAccountDebt
625         //
626         // example:
627         // - nonCollatInterestRate = 0.1
628         // - _timePast              = 3200
629         // - _currentAccountDebt    = 100
630         //
631         // _accountInterest        = 0.1 * 3200 * 100
632         //                          = 32000
633         _accountInterest =
634             (getNonCollatInterestRate(_account, _token, moneyMarketDs) * _timePast *
635             _currentAccountDebt) /
636             1e18;
637
638         // update non collat debt states
639         _newNonCollatDebtValue = _currentAccountDebt + _accountInterest;
640         // 1. account debt
641         moneyMarketDs.nonCollatAccountDebtValues[_account].addOrUpdate(_token,
642             _newNonCollatDebtValue);
643         // 2. token debt
644         moneyMarketDs.nonCollatTokenDebtValues[_token].addOrUpdate(_account,
645             _newNonCollatDebtValue);
646
647         // accumulate total non collat interest
648         _totalNonCollatInterest += _accountInterest;
649         unchecked {
650             ++_i;
651         }
652     }
653 }
```

However, there is no cap on the amount of non-collateral accounts, and if the amount of non-collateral is large enough, it can cause the gas to exceed the limit, resulting in a transaction being reverted.

The `accrueNonCollatInterest()` function is called by internal functions related to all processes. As a result, the user will be unable to perform any action on the platform.

5.5.2. Remediation

In the current design, Inspex suggests adding a cap for the amount of non-collateral accounts to mitigate the issue of exceeding the gas limit. For example:

Set up a cap for non-collateral accounts for 1000 accounts. Adding the `nonCollateCount` for counting the current amount of non collateral accounts.

LibMoneyMarket01.sol

```

125 mapping(address => LibDoublyLinkedList.List) nonCollatAccountDebtValues; //
    account => list token debt
126 mapping(address => LibDoublyLinkedList.List) nonCollatTokenDebtValues; // token
    => debt of each account
127 mapping(address => ProtocolConfig) protocolConfigs; // account =>
    ProtocolConfig
128 mapping(address => mapping(address => IInterestRateModel))
    nonCollatInterestModels; // [account][token] => non-collat interest model
129 mapping(address => bool) nonCollatBorrowerOk; // can this address do non collat
    borrow
130 uint256 nonCollatCount;
```

Validate that the `moneyMarketDs.nonCollatCount` is less than 1000.

AdminFacet.sol

```

206 function setNonCollatBorrowerOk(address _borrower, bool _isOk) external
    onlyOwner {
207     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
    LibMoneyMarket01.moneyMarketDiamondStorage();
208     if (_isOk == moneyMarketDs.nonCollatBorrowerOk[_borrower]){
209         return;
210     }
211     require(moneyMarketDs.nonCollatCount < 1000, "Non Collateral Account Exceed
    Capacity");
212     moneyMarketDs.nonCollatBorrowerOk[_borrower] = _isOk;
213     if (_isOk){
214         moneyMarketDs.nonCollatCount += 1;
215     } else{
216         moneyMarketDs.nonCollatCount -= 1;
217     }
218     emit LogsetNonCollatBorrowerOk(_borrower, _isOk);
219 }
```

5.6. Lack of Bad Debt Token Recovery after Write-Off

ID	IDX-006
Target	AdminFacet
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Medium</p> <p>Impact: Medium The borrower is paying more interest than expected, as the <code>writeOffSubAccountsDebt()</code> function decreases the debt of the protocol without increasing the protocol reserve. This reserve is used to calculate the utilization and interest rate for borrowers and the lender may be unable to withdraw their <code>_underlyingToken</code> due to the token being insufficient.</p> <p>Likelihood: Medium It is unlikely that the protocol owner will reset the user debt and will not top up to recover the debt amount.</p>
Status	<p>Resolved</p> <p>The Alpaca Finance team has resolved this issue by distributing losses equally among all lenders, resulting in a lower value of the lent token's shares than normal, which may be less than the principal value. This issue has been fixed in commit <code>48a6bd6491ff730ba5b596cdc073f01fc8bcc53c</code>.</p> <p>However, the bad debt will be recovered after the Governance Committee votes to allocate some of protocol's revenue to relieve the lenders' losses.</p>

5.6.1. Description

The `writeOffSubAccountsDebt()` function is used to reset the outstanding debt of the subaccount to zero. To reset the debt the function calls the `removeOverCollatDebtFromSubAccount()` function in line 537.

AdminFacet.sol

```

505 function writeOffSubAccountsDebt(WriteOffSubAccountDebtInput[] calldata
    _inputs) external onlyOwner {
506     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
    LibMoneyMarket01.moneyMarketDiamondStorage();
507
508     uint256 _length = _inputs.length;
509
510     address _token;
511     address _account;

```

```
512     address _subAccount;
513     uint256 _shareToRemove;
514     uint256 _amountToRemove;
515
516     for (uint256 i; i < _length; ) {
517         _token = _inputs[i].token;
518         _account = _inputs[i].account;
519         _subAccount = LibMoneyMarket01.getSubAccount(_account,
520             _inputs[i].subAccountId);
521
522         // Revert if the subAccount still have collateral left to be liquidated
523         if (moneyMarketDs.subAccountCollats[_subAccount].size != 0) {
524             revert AdminFacet_SubAccountHealthy(_subAccount);
525         }
526
527         // Accrue interest for token so debt share calculation would be correct
528         LibMoneyMarket01.accrueInterest(_token, moneyMarketDs);
529
530         // Get remaining debts of the token under subAccount
531         (_shareToRemove, _amountToRemove) =
532             LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
533                 _subAccount,
534                 _token,
535                 moneyMarketDs
536             );
537
538         // Reset debts of the token under subAccount
539         LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
540             _account,
541             _subAccount,
542             _token,
543             _shareToRemove,
544             _amountToRemove,
545             moneyMarketDs
546         );
547
548         emit LogWriteOffSubAccountDebt(_subAccount, _token, _shareToRemove,
549             _amountToRemove);
550
551         unchecked {
552             ++i;
553         }
554     }
555 }
```

The `removeOverCollatDebtFromSubAccount()` function decreases the debt of the subaccount and the repay token.

LibMoneyMarket01.sol

```

923 function removeOverCollatDebtFromSubAccount(
924     address _account,
925     address _subAccount,
926     address _repayToken,
927     uint256 _debtShareToRemove,
928     uint256 _debtValueToRemove,
929     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs
930 ) internal {
931     // get current debt share of a token
932     uint256 _currentDebtShare =
moneyMarketDs.subAccountDebtShares[_subAccount].getAmount(_repayToken);
933
934     // update states
935     // 1. update sub account debt share
936     moneyMarketDs.subAccountDebtShares[_subAccount].updateOrRemove(_repayToken,
currentDebtShare - _debtShareToRemove);
937     // 2. update total over collat debt share of a token in money market
938     moneyMarketDs.overCollatDebtShares[_repayToken] -= _debtShareToRemove;
939     // 3. update total over collat debt value of a token in money market
940     moneyMarketDs.overCollatDebtValues[_repayToken] -= _debtValueToRemove;
941     // 4. update total debt value of a token in money market
942     moneyMarketDs.globalDebts[_repayToken] -= _debtValueToRemove;
943
944     // withdraw debt token from miniFL
945     IMiniFL _miniFL = moneyMarketDs.miniFL;
946     address _debtToken = moneyMarketDs.tokenToDebtTokens[_repayToken];
947     _miniFL.withdraw(_account, moneyMarketDs.miniFLPoolIds[_debtToken],
_debtShareToRemove);
948
949     // burn debt token
950     IDebtToken(_debtToken).burn(address(this), _debtShareToRemove);
951
952     emit LogRemoveDebt(_account, _subAccount, _repayToken, _debtShareToRemove,
_debtValueToRemove);
953 }

```

After writing off the debt without increasing the reserve, the borrowing and lending interest rates will be higher than they should be due to the interest using the `globalDebts` state and the floating balance of the protocol.

LibMoneyMarket01.sol

```

496 function getOverCollatInterestRate(address _token, MoneyMarketDiamondStorage
storage moneyMarketDs)
497     internal
498     view

```

```

499     returns (uint256 _interestRate)
500 {
501     // get interest model of a token
502     IInterestRateModel _interestModel = moneyMarketDs.interestModels[_token];
503     // return 0 if interest model does not exist
504     // otherwise, return interest rate from interest model
505     if (address(_interestModel) == address(0)) {
506         return 0;
507     }
508     _interestRate = _interestModel.getInterestRate(
509         moneyMarketDs.globalDebts[_token],
510         getFloatingBalance(_token, moneyMarketDs)
511     );
512 }
513

```

TripleSlopeModel6.sol

```

17 function getInterestRate(uint256 debt, uint256 floating) external pure returns
   (uint256) {
18     if (debt == 0 && floating == 0) return 0;
19
20     uint256 total = debt + floating;
21     uint256 utilization = (debt * 100e18) / total;
22     if (utilization < CEIL_SLOPE_1) {
23         // Less than 85% utilization - 0%-17.5% APY
24         return (utilization * MAX_INTEREST_SLOPE_1) / (CEIL_SLOPE_1) / 365 days;
25     } else if (utilization < CEIL_SLOPE_2) {
26         // Between 85% and 90% - 17.5% APY
27         return uint256(MAX_INTEREST_SLOPE_2) / 365 days;
28     } else if (utilization < CEIL_SLOPE_3) {
29         // Between 90% and 100% - 17.5%-150% APY
30         return
31             (MAX_INTEREST_SLOPE_2 +
32              ((utilization - CEIL_SLOPE_2) * (MAX_INTEREST_SLOPE_3 -
33              MAX_INTEREST_SLOPE_2))) /
34             (CEIL_SLOPE_3 - CEIL_SLOPE_2)) / 365 days;
35     } else {
36         // Not possible, but just in case - 150% APY
37         return MAX_INTEREST_SLOPE_3 / 365 days;
38     }
39 }

```

The following scenario of write-off without top-up shows that the utilization state is decreased lower than the top-up, which means the borrower has to repay a higher amount and the lender receives a higher amount of interest.

State	Before Write-Off	Write-Off with Top-Up	Write-Off without Top-Up
floating	10	10+10=20	10
debt	90	90-10=80	90-10=80
total	100	100	90
utilization	$90 \times 100 / 100 = 90$	$80 \times 100 / 100 = 80$	$80 \times 100 / 90 = 88.88$

Furthermore, if the lender who lent the written-off token wants to withdraw before the platform owner calls the `topUpTokenReserve()` function to top up the reserve token, the lender may be unable to withdraw their tokens due to insufficient tokens.

AdminFacet.sol

```

557 function topUpTokenReserve(address _token, uint256 _amount) external onlyOwner
558 {
559     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
560     LibMoneyMarket01.moneyMarketDiamondStorage();
561     // Prevent topup token that didn't have market
562     if (moneyMarketDs.tokenToIbTokens[_token] == address(0)) revert
563     AdminFacet_InvalidToken(_token);
564     // Allow topup for token that has fee on transfer
565     uint256 _actualAmountReceived = LibMoneyMarket01.unsafePullTokens(_token,
566     msg.sender, _amount);
567     moneyMarketDs.reserves[_token] += _actualAmountReceived;
568     emit LogTopUpTokenReserve(_token, _actualAmountReceived);
569 }

```

5.6.2. Remediation

Inspex suggests modifying the `topUpTokenReserve()` function visibility to internal and using it inside `writeOffSubAccountsDebt()` function to recover the reserve immediately (line 546).

AdminFacet.sol

```

557 function topUpTokenReserve(address _token, uint256 _amount) internal {
558     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
559     LibMoneyMarket01.moneyMarketDiamondStorage();
560     // Prevent topup token that didn't have market
561     if (moneyMarketDs.tokenToIbTokens[_token] == address(0)) revert
562     AdminFacet_InvalidToken(_token);

```

```

563 // Allow topup for token that has fee on transfer
564 uint256 _actualAmountReceived = LibMoneyMarket01.unsafePullTokens(_token,
msg.sender, _amount);
565 moneyMarketDs.reserves[_token] += _actualAmountReceived;
566
567 emit LogTopUpTokenReserve(_token, _actualAmountReceived);
568 }

```

AdminFacet.sol

```

505 function writeOffSubAccountsDebt(WriteOffSubAccountDebtInput[] calldata
_inputs) external onlyOwner {
506     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
LibMoneyMarket01.moneyMarketDiamondStorage();
507
508     uint256 _length = _inputs.length;
509
510     address _token;
511     address _account;
512     address _subAccount;
513     uint256 _shareToRemove;
514     uint256 _amountToRemove;
515
516     for (uint256 i; i < _length; ) {
517         _token = _inputs[i].token;
518         _account = _inputs[i].account;
519         _subAccount = LibMoneyMarket01.getSubAccount(_account,
_inputs[i].subAccountId);
520
521         // Revert if the subAccount still have collateral left to be liquidated
522         if (moneyMarketDs.subAccountCollats[_subAccount].size != 0) {
523             revert AdminFacet_SubAccountHealthy(_subAccount);
524         }
525
526         // Accrue interest for token so debt share calculation would be correct
527         LibMoneyMarket01.accrueInterest(_token, moneyMarketDs);
528
529         // Get remaining debts of the token under subAccount
530         (_shareToRemove, _amountToRemove) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
531             _subAccount,
532             _token,
533             moneyMarketDs
534         );
535
536         // Reset debts of the token under subAccount
537         LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
538             _account,

```

```
539     _subAccount,  
540     _token,  
541     _shareToRemove,  
542     _amountToRemove,  
543     moneyMarketDs  
544 );  
545  
546     topUpTokenReserve(_token, _amountToRemove);  
547  
548     emit LogWriteOffSubAccountDebt(_subAccount, _token, _shareToRemove,  
549     _amountToRemove);  
549  
550     unchecked {  
551         ++i;  
552     }  
553 }  
554 }
```

5.7. Unable to Fully Repurchase Debt Share on Borrow Position

ID	IDX-007
Target	LiquidationFacet
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Medium</p> <p>Impact: Medium The borrowed position cannot be fully repurchased, leaving 1 Wei on that subaccount even though the collateral for that debt is fully bought out. This makes the platform accumulate the <code>globalDebt</code> state value and the borrower loses 1 Wei of the collateral, resulting in extra interest for the lenders and the borrowers.</p> <p>Likelihood: Medium This issue will occur when there are multiple active borrowed positions on the platform and the repurchasers can repurchase the debt account with the full amount of the <code>debtValue</code>.</p>
Status	<p>Resolved</p> <p>The Alpaca Finance team has resolved this issue by implementing the rounding up condition for the <code>repayAmountWithFee</code> when the user repurchasing the entire position in commit <code>48a6bd6491ff730ba5b596cdc073f01fc8bcc53c</code>.</p>

5.7.1. Description

When there are multiple available borrowed positions in the platform, the precision loss resulting from calculating the `debtShare` and `debtValue` can occur due to the nature of the EVM architecture. Given that, when the repurchasers can repurchase the borrowed position with the full collateral amount, the `debtShare` and `debtValue` can be left with 1 Wei.

This is because when the repurchasing occurs, the smart contract gets the debt (`_currentDebtAmount`) of that subaccount at line 178. After that, the `_currentDebtAmount` value is used to find the `_actualRepayAmountWithoutFee` value.

LiquidationFacet.sol

177	{
178	(, uint256 _currentDebtAmount) =
	LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
179	_vars.subAccount,
180	_repayToken,
181	moneyMarketDs
182);

```

183     uint256 _maxAmountRepurchaseable = (_currentDebtAmount *
184         (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS)) /
LibMoneyMarket01.MAX_BPS;
185
186     if (_desiredRepayAmount > _maxAmountRepurchaseable) {
187         _vars.repayAmountWithFee = _maxAmountRepurchaseable;
188         _vars.repayAmountWithoutFee = _currentDebtAmount;
189     } else {
190         _vars.repayAmountWithFee = _desiredRepayAmount;
191         _vars.repayAmountWithoutFee =
192             (_desiredRepayAmount * LibMoneyMarket01.MAX_BPS) /
193             (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS);
194     }
195
196     _vars.repurchaseFeeToProtocol = _vars.repayAmountWithFee -
_vars.repayAmountWithoutFee;
197 }

```

The `_vars.repayAmountWithFee` is later used to calculate the response share through the `LibShareUtil.valueToShare()` function at line 258.

LiquidationFacet.sol

```

246     // Remove subAccount debt
247     uint256 _actualRepayAmountWithoutFee = LibMoneyMarket01.unsafePullTokens(
248         _repayToken,
249         msg.sender,
250         _vars.repayAmountWithFee
251     ) - _vars.repurchaseFeeToProtocol;
252
253     // Remove subAccount debt
254     LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
255         _account,
256         _vars.subAccount,
257         _repayToken,
258         LibShareUtil.valueToShare(
259             _actualRepayAmountWithoutFee,
260             moneyMarketDs.overCollatDebtShares[_repayToken],
261             moneyMarketDs.overCollatDebtValues[_repayToken]
262         ),
263         _actualRepayAmountWithoutFee,
264         moneyMarketDs
265     );
266     // need to call removeCollat which might withdraw from miniFL to be able to
transfer to repurchaser
267     LibMoneyMarket01.removeCollatFromSubAccount(
268         _account,
269         _vars.subAccount,

```

```

270     _collatToken,
271     _collatAmountOut,
272     moneyMarketDs
273 );

```

This value will then be used to remove the share, debt, and collateral from that subaccount through the `removeOverCollatDebtFromSubAccount()` function.

LiquidationFacet.sol

```

923 function removeOverCollatDebtFromSubAccount(
924     address _account,
925     address _subAccount,
926     address _repayToken,
927     uint256 _debtShareToRemove,
928     uint256 _debtValueToRemove,
929     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs
930 ) internal {
931     // get current debt share of a token
932     uint256 _currentDebtShare =
moneyMarketDs.subAccountDebtShares[_subAccount].getAmount(_repayToken);
933
934     // update states
935     // 1. update sub account debt share
936     moneyMarketDs.subAccountDebtShares[_subAccount].updateOrRemove(_repayToken,
_currentDebtShare - _debtShareToRemove);
937     // 2. update total over collat debt share of a token in money market
938     moneyMarketDs.overCollatDebtShares[_repayToken] -= _debtShareToRemove;
939     // 3. update total over collat debt value of a token in money market
940     moneyMarketDs.overCollatDebtValues[_repayToken] -= _debtValueToRemove;
941     // 4. update total debt value of a token in money market
942     moneyMarketDs.globalDebts[_repayToken] -= _debtValueToRemove;
943
944     // withdraw debt token from miniFL
945     IMiniFL _miniFL = moneyMarketDs.miniFL;
946     address _debtToken = moneyMarketDs.tokenToDebtTokens[_repayToken];
947     _miniFL.withdraw(_account, moneyMarketDs.miniFLPoolIds[_debtToken],
_debtShareToRemove);
948
949     // burn debt token
950     IDebtToken(_debtToken).burn(address(this), _debtShareToRemove);
951
952     emit LogRemoveDebt(_account, _subAccount, _repayToken, _debtShareToRemove,
_debtValueToRemove);
953 }

```

As a result, there will be left with 1 extra Wei for `debtShare` in that subaccount.

5.7.2. Remediation

Inspex suggests adding a condition to verify that there is no precision loss for the full repurchasing in the `repurchase()` function, so there will be an extra amount charged to the repurchaser.

Add `isPurchaseAll` state to the struct.

LiquidationFacet.sol

```

47 struct RepurchaseLocalVars {
48     address subAccount;
49     uint256 totalBorrowingPower;
50     uint256 usedBorrowingPower;
51     uint256 repayAmountWithFee;
52     uint256 repurchaseFeeToProtocol;
53     uint256 repurchaseRewardBps;
54     uint256 repayAmountWithoutFee;
55     uint256 repayTokenPrice;
56     bool isPurchaseAll;
57 }

```

Mark the `isPurchaseAll` state when this repurchasing is a full repurchase.

LiquidationFacet.sol

```

178 {
179     (, uint256 _currentDebtAmount) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
180         _vars.subAccount,
181         _repayToken,
182         moneyMarketDs
183     );
184     uint256 _maxAmountRepurchaseable = (_currentDebtAmount *
185         (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS)) /
LibMoneyMarket01.MAX_BPS;
186
187     if (_desiredRepayAmount > _maxAmountRepurchaseable) {
188         _vars.repayAmountWithFee = _maxAmountRepurchaseable;
189         _vars.repayAmountWithoutFee = _currentDebtAmount;
190         _vars.isPurchaseAll = true;
191     } else {
192         _vars.repayAmountWithFee = _desiredRepayAmount;
193         _vars.repayAmountWithoutFee =
194             (_desiredRepayAmount * LibMoneyMarket01.MAX_BPS) /
195             (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS);
196         _vars.isPurchaseAll = false;
197     }
198
199     _vars.repurchaseFeeToProtocol = _vars.repayAmountWithFee -

```

```
200 _vars.repayAmountWithoutFee;  
    }
```

Add a condition for full repurchasing to cover the precision loss case.

LiquidationFacet.sol

```
249 uint256 _actualRepayAmountWithoutFee;  
250 {  
251     if (_vars.isPurchaseAll){  
252         uint256 _repayAmountWithoutFee = _vars.repayAmountWithFee -  
_vars.repurchaseFeeToProtocol;  
253         uint256 _actualRepayAmountAsShare = LibShareUtil.valueToShare(  
254             _repayAmountWithoutFee,  
255             moneyMarketDs.overCollatDebtShares[_repayToken],  
256             moneyMarketDs.overCollatDebtValues[_repayToken]  
257         );  
258         uint256 _expectRepayAmountAsShare = LibShareUtil.valueToShareRoundingUp(  
259             _repayAmountWithoutFee,  
260             moneyMarketDs.overCollatDebtShares[_repayToken],  
261             moneyMarketDs.overCollatDebtValues[_repayToken]  
262         );  
263         if (_actualRepayAmountAsShare + 1 == _expectRepayAmountAsShare) {  
264             _vars.repayAmountWithFee += 1;  
265         }  
266     }  
267     _actualRepayAmountWithoutFee = LibMoneyMarket01.unsafePullTokens(  
268         _repayToken,  
269         msg.sender,  
270         _vars.repayAmountWithFee  
271     ) - _vars.repurchaseFeeToProtocol;  
272 }
```


5.8. Design Flaw in Supporting Deflationary Token

ID	IDX-008
Target	BorrowFacet LiquidationFacet NonCollatBorrowFacet MiniFL
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Low</p> <p>Impact: Medium</p> <p>With the current implementation that opens up the scenario for the deflationary token, the deflationary token can interrupt the business design since most of the functionalities rely on the token transfer which is where the fee occurs. For example, if the position is repurchasable, the amount that the repurchaser received will be less than what it should be, resulting in less incentive for the repurchaser which increases the chance of bad debt for the platform.</p> <p>Likelihood: Low</p> <p>The deflationary token can only be used in the protocol after the platform has opened the market for it. Furthermore, the deflationary tokens cannot be lent as the platform liquidity due to the logic of the <code>deposit()</code> function (applying the <code>pullExactTokens()</code> function). Hence, it is unlikely that the platform will allow the deflationary token to be used.</p>
Status	<p>Resolved</p> <p>The Alpaca Finance team has clarified that they will not support the deflationary token.</p>

5.8.1. Description

With the current implementation, the platform can support many types of tokens to be used in the protocol such as the deflationary token type as shown in the `unsafePullTokens()` function, which is used in multiple contracts.

LibMoneyMarket01.sol

```

1113  /// @dev SafeTransferFrom that return actual amount received
1114  /// @param _token The token address
1115  /// @param _from The address to pull token from
1116  /// @param _amount The amount to pull
1117  /// @return _actualAmountReceived The actual amount received
1118  function unsafePullTokens(
1119      address _token,
1120      address _from,

```

```

1121     uint256 _amount
1122 ) internal returns (uint256 _actualAmountReceived) {
1123     // get the token balance of money market before transfer
1124     uint256 _balanceBefore = IERC20(_token).balanceOf(address(this));
1125     // transfer token from _from to money market
1126     IERC20(_token).safeTransferFrom(_from, address(this), _amount);
1127     // return actual amount received = balance after transfer - balance before
    transfer
1128     _actualAmountReceived = IERC20(_token).balanceOf(address(this)) -
    _balanceBefore;
1129 }

```

While the `pullExactTokens()` function supports the common ERC20 token.

LibMoneyMarket01.sol

```

1103 /// @dev SafeTransferFrom that revert when not receiving full amount (have fee
    on transfer)
1104 /// @param _token The token address
1105 /// @param _from The address to pull token from
1106 /// @param _amount The amount to pull
1107 function pullExactTokens(
1108     address _token,
1109     address _from,
1110     uint256 _amount
1111 ) internal {
1112     // get the token balance of money market before transfer
1113     uint256 _balanceBefore = IERC20(_token).balanceOf(address(this));
1114     // transfer token from _from to money market
1115     IERC20(_token).safeTransferFrom(_from, address(this), _amount);
1116     // check if the token balance of money market is increased by _amount
1117     // if fee on transfer tokens is not supported this will revert
1118     if (IERC20(_token).balanceOf(address(this)) - _balanceBefore != _amount) {
1119         revert LibMoneyMarket01_FeeOnTransferTokensNotSupported();
1120     }
1121 }

```

However, the deflationary token is not properly supported and it may lead to many problems if the platform tries to use it in the protocol, for example:

Scenario 1: Miscalculation when repurchasing

In the repurchasing process, the `repurchase()` function is used to purchase the collateral at a cheaper price by providing the debt token to the platform. The `_collatAmountOut` value is then calculated, which determines the amount of collateral that can be purchased for the repurchaser, after which the collateral will be withdrawn from the `MiniFL` contract.

LiquidationFacet.sol

```
95 function repurchase(  
96     address _account,  
97     uint256 _subAccountId,  
98     address _repayToken,  
99     address _collatToken,  
100    uint256 _desiredRepayAmount  
101 ) external nonReentrant returns (uint256 _collatAmountOut) {  
102     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =  
103     LibMoneyMarket01.moneyMarketDiamondStorage();  
104     // We only allow EOA or whitelisted contract to repurchase  
105     // Revert if caller is contract that is not whitelisted  
106     // `msg.sender != tx.origin` means that `msg.sender` is contract  
107     if (msg.sender != tx.origin && !moneyMarketDs.repurchasersOk[msg.sender]) {  
108         revert LiquidationFacet_Unauthorized();  
109     }  
110  
111     RepurchaseLocalVars memory _vars;  
112  
113     _vars.subAccount = LibMoneyMarket01.getSubAccount(_account, _subAccountId);  
114  
115     // Accrue all debt tokens under subaccount  
116     // Because used borrowing power is calculated from all debt token of the  
117     subaccount  
118     LibMoneyMarket01.accrueBorrowedPositionsOf(_vars.subAccount, moneyMarketDs);  
119  
120     _vars.totalBorrowingPower =  
121     LibMoneyMarket01.getTotalBorrowingPower(_vars.subAccount, moneyMarketDs);  
122     (_vars.usedBorrowingPower, ) =  
123     LibMoneyMarket01.getTotalUsedBorrowingPower(_vars.subAccount, moneyMarketDs);  
124     // Revert if position is not repurchasable (borrowingPower /  
125     usedBorrowingPower >= 1)  
126     if (_vars.totalBorrowingPower >= _vars.usedBorrowingPower) {  
127         revert LiquidationFacet_Healthy();  
128     }  
129  
130     // Cap repurchase amount if needed and calculate fee  
131     // Fee is calculated from repaid amount  
132     //  
133     // formulas:  
134     //     maxAmountRepurchaseable = currentDebt * (1 + fee)  
135     //     repurchaseFee            = repayAmountWithFee - repayAmountWithoutFee  
136     //     if desiredRepayAmount > maxAmountRepurchaseable  
137     //         repayAmountWithFee    = maxAmountRepurchaseable  
138     //         repayAmountWithoutFee = currentDebt  
139     //     else  
140     //         repayAmountWithFee    = desiredRepayAmount
```

```

137 //      repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
138 //
139 // calculation example:
140 //      assume 1 eth = 2000 USD, 10% repurchase fee, ignore
collat,borrowingFactor, no premium
141 //      collateral: 2000 USDC
142 //      debt      : 1 eth
143 //      maxAmountRepurchaseable = currentDebt * (1 + fee)
144 //                               = 1 * (1 + 0.1) = 1.1 eth
145 //
146 //      ex 1: desiredRepayAmount > maxAmountRepurchaseable
147 //      input : desiredRepayAmount = 1.2 eth
148 //      repayAmountWithFee      = maxAmountRepurchaseable = 1.1 eth
149 //      repayAmountWithoutFee = currentDebt = 1 eth
150 //      repurchaseFee           = repayAmountWithFee - repayAmountWithoutFee
151 //                               = 1.1 - 1 = 0.1 eth
152 //
153 //      ex 2: desiredRepayAmount < currentDebt
154 //      input : desiredRepayAmount = 0.9 eth
155 //      repayAmountWithFee      = desiredRepayAmount = 0.9 eth
156 //      repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
157 //                               = 0.9 / (1 + 0.1) = 0.818181... eth
158 //      repurchaseFee           = repayAmountWithFee - repayAmountWithoutFee
159 //                               = 0.9 - 0.818181... = 0.0818181... eth
160 //
161 //      ex 3: desiredRepayAmount == currentDebt
162 //      input : desiredRepayAmount = 1 eth
163 //      repayAmountWithFee      = desiredRepayAmount = 1 eth
164 //      repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
165 //                               = 1 / (1 + 0.1) = 0.909090... eth
166 //      repurchaseFee           = repayAmountWithFee - repayAmountWithoutFee
167 //                               = 1 - 0.909090... = 0.0909090... eth
168 //
169 //      ex 4: currentDebt < desiredRepayAmount < maxAmountRepurchaseable
170 //      input : desiredRepayAmount = 1.05 eth
171 //      repayAmountWithFee = desiredRepayAmount = 1.05 eth
172 //      repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
173 //                               = 1.05 / (1 + 0.1) = 0.9545454... eth
174 //      repurchaseFee           = repayAmountWithFee - repayAmountWithoutFee
175 //                               = 1.05 - 0.9545454... = 0.09545454... eth
176 //
177 {
178     (, uint256 _currentDebtAmount) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
179         _vars.subAccount,
180         _repayToken,
181         moneyMarketDs

```

```
182     );
183     uint256 _maxAmountRepurchaseable = (_currentDebtAmount *
184         (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS)) /
LibMoneyMarket01.MAX_BPS;
185
186     if (_desiredRepayAmount > _maxAmountRepurchaseable) {
187         _vars.repayAmountWithFee = _maxAmountRepurchaseable;
188         _vars.repayAmountWithoutFee = _currentDebtAmount;
189     } else {
190         _vars.repayAmountWithFee = _desiredRepayAmount;
191         _vars.repayAmountWithoutFee =
192             (_desiredRepayAmount * LibMoneyMarket01.MAX_BPS) /
193             (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS);
194     }
195
196     _vars.repurchaseFeeToProtocol = _vars.repayAmountWithFee -
_vars.repayAmountWithoutFee;
197 }
198
199 _vars.repayTokenPrice = LibMoneyMarket01.getPriceUSD(_repayToken,
moneyMarketDs);
200
201 // Revert if repayment exceeds threshold (repayment > maxLiquidateThreshold *
usedBorrowingPower)
202 _validateBorrowingPower(_repayToken, _vars.repayAmountWithoutFee,
_vars.usedBorrowingPower, moneyMarketDs);
203
204 // Get dynamic repurchase reward to further incentivize repurchase
205 _vars.repurchaseRewardBps = moneyMarketDs.repurchaseRewardModel.getFeeBps(
206     _vars.totalBorrowingPower,
207     _vars.usedBorrowingPower
208 );
209
210 // Calculate payout for repurchaser (collateral with premium)
211 {
212     uint256 _collatTokenPrice = LibMoneyMarket01.getPriceUSD(_collatToken,
moneyMarketDs);
213
214     uint256 _repayTokenPriceWithPremium = (_vars.repayTokenPrice *
215         (LibMoneyMarket01.MAX_BPS + _vars.repurchaseRewardBps)) /
LibMoneyMarket01.MAX_BPS;
216
217     // collatAmountOut = repayAmount * repayTokenPriceWithPremium /
collatTokenPrice
218     _collatAmountOut =
219         (_vars.repayAmountWithFee *
220         _repayTokenPriceWithPremium *
```

```

221         moneyMarketDs.tokenConfigs[_repayToken].to18ConversionFactor) /
222         (_collatTokenPrice *
moneyMarketDs.tokenConfigs[_collatToken].to18ConversionFactor);
223
224     // revert if subAccount collat is not enough to cover desired repay amount
225     // this could happen when there are multiple small collat and one large
debt
226     // ex. assume 1 eth = 2000 USD, no repurchase fee or premium, ignore
collat,borrowingFactor
227     //     collateral : 1000 USDT, 1000 USDC
228     //     debt       : 1 eth
229     //     input      : desiredRepayAmount = 0.6 eth, collatToken = USDC
230     //     collatAmountOut = repayAmount * repayTokenPrice / collatTokenPrice
231     //                   = 0.6 * 2000 / 1 = 1200 USDC
232     //     this should revert since there is not enough USDC collateral to be
repurchased
233     if (_collatAmountOut >
moneyMarketDs.subAccountCollats[_vars.subAccount].getAmount(_collatToken)) {
234         revert LiquidationFacet_InsufficientAmount();
235     }
236 }
237
238 ////////////////////////////////////////////////////
239 // EFFECTS & INTERACTIONS
240 ////////////////////////////////////////////////////
241
242 // Transfer repay token in
243 // In case of token with fee on transfer, debt would be repaid by amount
after transfer fee
244 // which won't be able to repurchase entire position
245 // repaidAmount = amountReceived - repurchaseFee
246 uint256 _actualRepayAmountWithoutFee = LibMoneyMarket01.unsafePullTokens(
247     _repayToken,
248     msg.sender,
249     _vars.repayAmountWithFee
250 ) - _vars.repurchaseFeeToProtocol;
251
252 // Remove subAccount debt
253 LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
254     _account,
255     _vars.subAccount,
256     _repayToken,
257     LibShareUtil.valueToShare(
258         _actualRepayAmountWithoutFee,
259         moneyMarketDs.overCollatDebtShares[_repayToken],
260         moneyMarketDs.overCollatDebtValues[_repayToken]
261     ),

```

```

262     _actualRepayAmountWithoutFee,
263     moneyMarketDs
264 );
265 // need to call removeCollat which might withdraw from miniFL to be able to
transfer to repurchaser
266 LibMoneyMarket01.removeCollatFromSubAccount(
267     _account,
268     _vars.subAccount,
269     _collatToken,
270     _collatAmountOut,
271     moneyMarketDs
272 );
273
274 // Increase reserves balance with repaid tokens
275 // Safe to use unchecked because _actualRepayAmountWithoutFee is derived from
balanceOf
276 unchecked {
277     moneyMarketDs.reserves[_repayToken] += _actualRepayAmountWithoutFee;
278 }
279
280 // Transfer collat token with premium back to repurchaser
281 IERC20(_collatToken).safeTransfer(msg.sender, _collatAmountOut);
282 // Transfer protocol's repurchase fee to treasury
283 IERC20(_repayToken).safeTransfer(moneyMarketDs.liquidationTreasury,
_vars.repurchaseFeeToProtocol);
284
285 emit LogRepurchase(
286     msg.sender,
287     _account,
288     _subAccountId,
289     _repayToken,
290     _collatToken,
291     _actualRepayAmountWithoutFee,
292     _collatAmountOut,
293     _vars.repurchaseFeeToProtocol,
294     (_collatAmountOut * _vars.repurchaseRewardBps) / LibMoneyMarket01.MAX_BPS
295 );
296 }

```

The withdrawn collateral will be transferred to the **MoneyMarketDiamond** contract and then to the `msg.sender`.

For example, if the `_collatAmountOut` is equal to 100, the deflationary token burn 5% of the amount during transfer, and the position can be fully repurchased (`maxLiquidateBps` is 10000), the repurchasing process will be as follows:

1. Assume that `_vars.repayAmountWithFee` is equal to 100 and `_vars.repurchaseFeeToProtocol` is

equal to 5.

2. Execute the `unsafePullTokens()` function, then the contract received only 95 tokens, and the `_actualRepayAmountWithoutFee` variable will be equal to 90, which is the received amount of tokens after being deducted by the protocol's fee.
3. Execute the `removeOverCollatDebtFromSubAccount()` function, assume that the value to share is now 1:1, and the debt of the borrower will only reduce by 95 instead of 100.
4. Execute the `removeCollatFromSubAccount()` function to withdraw 100 tokens, according to the value of `_collatAmountOut`, from the `MiniFL` contract.
5. The received amount from `MiniFL` will be 100 tokens due to the `_collatAmountOut` value.
6. The contract transfers 100 tokens of collateral, according to the value of `_collatAmountOut`, to the repurchaser, even though the contract only received 90 tokens from step (2).

As a result, the `MoneyMarketDiamond` contract will receive fewer debt tokens than the actual amount that was used to convert to the `_collatAmountOut` before. Therefore, bad debt will occur, resulting in a loss of money for the lenders.

Scenario 2: More transfers, more fees collected

Given that the `DFT` deflationary token will burn 5% of the amount during transfer.

In the event that the platform perfectly supports deflationary tokens and allows them to be used in the protocol such as using the `unsafePullTokens()` function instead of the `pullExactTokens()` function, the user deposits 100 tokens of `DFT` by calling `deposit()` function in the `MoneyMarketAccountManager` contract, and the token will be transferred as follows:

1. 100 tokens will be transferred from the user to the `MoneyMarketAccountManager` contract, but the contract will receive only 95 tokens.
2. 95 tokens will be transferred from the `MoneyMarketAccountManager` contract to the `MoneyMarketDiamond` contract, but the contract will receive only 90.25 tokens.

As a result, the fee is collected twice for the depositing process, and it will be collected three times for the adding/removing collateral process. The more transfers, the more fees that will be collected.

5.8.2. Remediation

Inspex suggests that it may not be advisable to support the deflationary token. The reason for this is that deflationary tokens typically reduce their total supply and received amount during transfer, which could lead to unintended consequences.

5.9. Inconsistent Interest Accrual Frequency

ID	IDX-009
Target	BorrowFacet LendFacet CollateralFacet LiquidationFacet NonCollatBorrowFacet
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	<p>Severity: Low</p> <p>Impact: Medium</p> <p>The interest accrued by the Alpaca protocol increases with every call of the <code>accrueInterest()</code> function. Therefore, a borrower and lender will pay or receive less interest if the function is called fewer times.</p> <p>Likelihood: Low</p> <p>This issue is unlikely to occur, since the platform implements the external <code>accrueInterest()</code> function in the <code>BorrowFacet</code> facet that enables the bot to call periodically. Moreover, the <code>accrueInterest()</code> is called internally for all functionalities that are related to the interest flow.</p>
Status	<p>Acknowledged</p> <p>The Alpaca Finance team has acknowledged this issue.</p>

5.9.1. Description

The Alpaca platform enables users to lend and borrow tokens. When the `accrueInterest()` function is called in the `LibMoneyMarket01` library, the interests for both lenders and borrowers will be distributed in the contract by the internal calculation.

LibMoneyMarket01.sol

```

558 function accrueInterest(address _token, MoneyMarketDiamondStorage storage
    moneyMarketDs) internal {
559     uint256 _lastAccrualTimestamp = moneyMarketDs.debtLastAccruedAt[_token];
560     // skip if interest has already been accrued within this block
561     if (block.timestamp > _lastAccrualTimestamp) {
562         // get a period of time since last accrual in seconds
563         uint256 _secondsSinceLastAccrual;
564         // safe to use unchecked
565         // because at this statement, block.timestamp is always greater than
        _lastAccrualTimestamp
    
```

```

566     unchecked {
567         _secondsSinceLastAccrual = block.timestamp - _lastAccrualTimestamp;
568     }
569     // accrue interest
570     uint256 _overCollatInterest = accrueOverCollatInterest(_token,
571 _secondsSinceLastAccrual, moneyMarketDs);
572     uint256 _nonCollatInterest = accrueNonCollatInterest(_token,
573 _secondsSinceLastAccrual, moneyMarketDs);
574
575     // update global debt
576     uint256 _totalInterest = _overCollatInterest + _nonCollatInterest;
577     moneyMarketDs.globalDebts[_token] += _totalInterest;
578
579     // update timestamp
580     moneyMarketDs.debtLastAccruedAt[_token] = block.timestamp;
581
582     // book protocol's revenue
583     // calculation:
584     // _protocolFee = (_totalInterest * lendingFeeBps) / MAX_BPS
585     //
586     // example:
587     // - _totalInterest = 1
588     // - lendingFeeBps = 1900
589     // - MAX_BPS      = 10000
590     //
591     // _protocolFee    = (1 * 1900) / 10000
592     //                  = 0.19
593     uint256 _protocolFee = (_totalInterest * moneyMarketDs.lendingFeeBps) /
594 MAX_BPS;
595     moneyMarketDs.protocolReserves[_token] += _protocolFee;
596
597     emit LogAccrueInterest(_token, _totalInterest, _protocolFee);
598 }
599 }

```

Every time the `accrueInterest()` function is called, the global debt increases cumulatively by an amount proportional to the time passed since the previous call. This amount is represented by the `_totalInterest` state, and reflects the new interest accrued from both lenders and borrowers.

Therefore, the frequency of the `accrueInterest()` function call is affected to the interest of borrower and lender.

For example, using the `FixedInterestRateModel` with an interest rate of 0.1% per second.

Scenario 1: Bob borrowed 1 \$WETH and only called the `accrueInterest()` function after 50 seconds.

1. Bob added collateral for 20 \$WETH

2. Bob borrowed 1 \$WETH
3. Bob checked the debt amount
 - Bob's \$WETH **_debtShare**: 1000000000000000000
 - Bob's \$WETH **_debtAmount**: 1000000000000000000
4. After 50 seconds had passed, the **accrueInterest()** function was called.
5. Bob's checked debt amount again
 - Bob's \$WETH **_debtShare**: 1000000000000000000
 - Bob's \$WETH **_debtAmount**: 1050000000000000000

Scenario 2: Bob borrowed 1 \$WETH and called the **accrueInterest()** function every 10 seconds, 5 times (for a total of 50 seconds).

1. Bob added collateral for 20 \$WETH
2. Bob borrowed 1 \$WETH
3. Bob checked the debt amount
 - Bob's \$WETH **_debtShare**: 1000000000000000000
 - Bob's \$WETH **_debtAmount**: 1000000000000000000
4. After 10 seconds had passed, the **accrueInterest()** function was called.
5. After 10 seconds had passed, the **accrueInterest()** function was called.
6. After 10 seconds had passed, the **accrueInterest()** function was called.
7. After 10 seconds had passed, the **accrueInterest()** function was called.
8. After 10 seconds had passed, the **accrueInterest()** function was called.
9. Bob checked debt amount again
 - Bob's \$WETH **_debtShare**: 1000000000000000000
 - Bob's \$WETH **_debtAmount**: 1052072264697058413

As shown in the results above, when the **accrueInterest()** function has not been called gradually, the compounding interest will be less than from what it should be according to the interest model that yields the interest for each second.

5.9.2. Remediation

Inspex suggests creating watchers to call the **accrueInterest()** function at intervals, such as once every hour for all tokens. This ensures that the interest calculation will compound at least once every hour.

5.10. Incorrect Logging Parameter

ID	IDX-010
Target	BorrowFacet AdminFacet NonCollatBorrowFacet LibMoneyMarket01
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	Severity: Info Impact: None Likelihood: None
Status	No Security Impact The Alpaca Finance team has clarified that the current logging will support multiple types of the <code>AccountManager</code> contract which will be implemented in the future.

5.10.1. Description

The information that was logged is invalid, which can lead to user misunderstandings.

For example, in the `BorrowFacet` facet, the `repay()` function is logging the function caller using the `msg.sender` as shown below in the following source code:

BorrowFacet.sol

```
93 function repay(  
94     address _account,  
95     uint256 _subAccountId,  
96     address _token,  
97     uint256 _debtShareToRepay  
98 ) external nonReentrant {  
99     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =  
100     LibMoneyMarket01.moneyMarketDiamondStorage();  
101     // This function should not be called from anyone  
102     // except account manager contract and will revert upon trying to do so  
103     LibMoneyMarket01.onlyAccountManager(moneyMarketDs);  
104  
105     address _subAccount = LibMoneyMarket01.getSubAccount(_account,  
106     _subAccountId);  
107     // accrue all debt tokens under subaccount
```

```
108 // because used borrowing power is calculated from all debt token of sub
account
109 LibMoneyMarket01 accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);
110
111 // Get the current debt amount and share of this token under the subaccount
112 // The current debt share will be used to cap the maximum that can be repaid
113 // The current debt amount will be used to check the minimum debt size after
repaid
114 (uint256 _currentDebtShare, uint256 _currentDebtAmount) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
115     _subAccount,
116     _token,
117     moneyMarketDs
118 );
119
120 // The debt share that can be repaid should not exceed the current debt share
121 // that the subaccount is holding
122 uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare,
_debtShareToRepay);
123
124 // caching these variables to save gas from multiple reads
125 uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
126 uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];
127
128 // Find the actual underlying amount that need to be pulled from the share
129 uint256 _actualAmountToRepay = LibShareUtil.shareToValue(_actualShareToRepay,
_cachedDebtValue, _cachedDebtShare);
130
131 // Pull the token from the account manager, the actual amount received will
be used for debt accounting
132 // In case somehow there's fee on transfer - which's might be introduced
after the token was lent
133 // Not reverting to ensure that repay transaction can be done even if there's
fee on transfer
134 _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender,
_actualAmountToRepay);
135
136 // Recalculate the debt share to remove in case there's fee on transfer
137 _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay,
_cachedDebtShare, _cachedDebtValue);
138
139 // Increase the reserve amount of the token as there's new physical token
coming in
140 moneyMarketDs.reserves[_token] += _actualAmountToRepay;
141
142 // Check and revert if the repay transaction will violate the business rule
143 // namely the debt size after repaid should be more than minimum debt size
```

```

144     _validateRepay(
145         _token,
146         _currentDebtShare,
147         _currentDebtAmount,
148         _actualShareToRepay,
149         _actualAmountToRepay,
150         moneyMarketDs
151     );
152
153     // Remove the debt share from this subaccount's accounting
154     // additionally, this library call will unstake the debt token
155     // from miniFL and burn the debt token
156     LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
157         _account,
158         _subAccount,
159         _token,
160         _actualShareToRepay,
161         _actualAmountToRepay,
162         moneyMarketDs
163     );
164
165     emit LogRepay(_account, _subAccountId, _token, msg.sender,
166         _actualAmountToRepay);
167 }

```

However, the `repay()` function can only be called by the `MoneyMarketAccountManager` contract, which will lead to all `LogRepay` events logging the `MoneyMarketAccountManager` address as the caller.

Furthermore, there is another instance of invalid logging, as shown in the following table:

Target	Event	Parameter	Remark
BorrowFacet.sol(L:165)	LogRepay()	_caller	The <code>_caller</code> parameter should be the address of the account that called the function in the <code>MoneyMarketAccountManager</code> contract
AdminFacet.sol(L:57)	LogWriteOffSubAccountDebt()	subAccount	The <code>account</code> parameter should be logged along with the <code>subAccount</code> parameter
NonCollatBorrowFacet.sol(L:21)	LogNonCollatBorrow()	_removeDebtAmount	The parameter's name should be declared according to its value,

			e.g., <code>_borrowAmount</code>
LibMoneyMarket01.sol(L:56)	LogRemoveCollateral()	-	The caller of this transaction should be logged, so the user can track why their collateral was removed
LibMoneyMarket01.sol(L:856)	LogAddCollateral()	<code>_caller</code>	The <code>_caller</code> parameter should be the address of the account that called the function in the MoneyMarketAccountManager contract
LibMoneyMarket01.sol(L:1063)	LogOverCollatBorrow()	<code>_account</code>	The <code>_account</code> variable should be used as the <code>_account</code> parameter instead of the <code>msg.sender</code> variable

The current implementation may confuse users who inspect the logs, as the logged information does not accurately reflect the function caller.

5.10.2. Remediation

Inspex suggests changing the logging to the valid value. For example, adding the `_caller` parameter to the `repay()` function.

BorrowFacet.sol

```

93 function repay(
94     address _account,
95     uint256 _subAccountId,
96     address _token,
97     uint256 _debtShareToRepay,
98     address _caller
99 ) external nonReentrant {
100     LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
101     LibMoneyMarket01.moneyMarketDiamondStorage();
102
103     // This function should not be called from anyone
104     // except account manager contract and will revert upon trying to do so
105     LibMoneyMarket01.onlyAccountManager(moneyMarketDs);
106
107     address _subAccount = LibMoneyMarket01.getSubAccount(_account,
108     _subAccountId);
109 
```

```
108 // accrue all debt tokens under subaccount
109 // because used borrowing power is calculated from all debt token of sub
account
110 LibMoneyMarket01.accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);
111
112 // Get the current debt amount and share of this token under the subaccount
113 // The current debt share will be used to cap the maximum that can be repaid
114 // The current debt amount will be used to check the minimum debt size after
repaid
115 (uint256 _currentDebtShare, uint256 _currentDebtAmount) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
116     _subAccount,
117     _token,
118     moneyMarketDs
119 );
120
121 // The debt share that can be repaid should not exceed the current debt share
122 // that the subaccount is holding
123 uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare,
_debtShareToRepay);
124
125 // caching these variables to save gas from multiple reads
126 uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
127 uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];
128
129 // Find the actual underlying amount that need to be pulled from the share
130 uint256 _actualAmountToRepay = LibShareUtil.shareToValue(_actualShareToRepay,
_cachedDebtValue, _cachedDebtShare);
131
132 // Pull the token from the account manager, the actual amount received will
be used for debt accounting
133 // In case somehow there's fee on transfer - which's might be introduced
after the token was lent
134 // Not reverting to ensure that repay transaction can be done even if there's
fee on transfer
135 _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender,
_actualAmountToRepay);
136
137 // Recalculate the debt share to remove in case there's fee on transfer
138 _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay,
_cachedDebtShare, _cachedDebtValue);
139
140 // Increase the reserve amount of the token as there's new physical token
coming in
141 moneyMarketDs.reserves[_token] += _actualAmountToRepay;
142
143 // Check and revert if the repay transaction will violate the business rule
```



```

144 // namely the debt size after repaid should be more than minimum debt size
145 _validateRepay(
146     _token,
147     _currentDebtShare,
148     _currentDebtAmount,
149     _actualShareToRepay,
150     _actualAmountToRepay,
151     moneyMarketDs
152 );
153
154 // Remove the debt share from this subaccount's accounting
155 // additionally, this library call will unstake the debt token
156 // from miniFL and burn the debt token
157 LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
158     _account,
159     _subAccount,
160     _token,
161     _actualShareToRepay,
162     _actualAmountToRepay,
163     moneyMarketDs
164 );
165
166 emit LogRepay(_account, _subAccountId, _token, _caller,
167     _actualAmountToRepay);
168 }

```

Thus, adding the `msg.sender` as the caller of the `repayFor()` function.

MoneyMarketAccountManager.sol

```

325 function repayFor(
326     address _account,
327     uint256 _subAccountId,
328     address _token,
329     uint256 _repayAmount,
330     uint256 _debtShareToRepay
331 ) external {
332     // cache the balance of token before proceeding
333     uint256 _amountBefore = IERC20(_token).balanceOf(address(this));
334
335     // Fund this contract from caller
336     // ignore the fact that there might be fee on transfer
337     IERC20(_token).safeTransferFrom(msg.sender, address(this), _repayAmount);
338
339     // Call repay by forwarding input _debtShareToRepay
340     // Money Market should deduct the fund as much as possible
341     // If there's excess amount left, transfer back to user
342     IERC20(_token).safeApprove(address(moneyMarket), _repayAmount);

```

```
343     moneyMarket.repay(_account, _subAccountId, _token, _debtShareToRepay,
344     msg.sender);
345     // Reset allowance as moneyMarket.repay() might not use all the allowance
346     IERC20(_token).safeApprove(address(moneyMarket), 0);
347     // Calculate the excess amount left in the contract
348     // This will revert if the input repay amount has lower value than
349     _debtShareToRepay
350     // And there's some token left in contract (can be done by inject token
351     directly to this contract)
352     uint256 _excessAmount = IERC20(_token).balanceOf(address(this)) -
353     _amountBefore;
354     if (_excessAmount != 0) {
355         IERC20(_token).safeTransfer(msg.sender, _excessAmount);
356     }
```

5.11. Inconsistent Usage of Function Caller Identifiers

ID	IDX-011
Target	DebtToken
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	Severity: Info Impact: None Likelihood: None
Status	Resolved The Alpaca Finance team has resolved this issue by using <code>msg.sender</code> instead of <code>_msgSender()</code> in commit <code>48a6bd6491ff730ba5b596cdc073f01fc8bcc53c</code> .

5.11.1. Description

In the `DebtToken` contract, the `transferFrom` function is used to transfer tokens from an allowed address to a destination address, as shown in the following source code:

DebtToken.sol

```
87 function transferFrom(  
88     address from,  
89     address to,  
90     uint256 amount  
91 ) public override returns (bool) {  
92     if (!(okHolders[msg.sender] && okHolders[from] && okHolders[to])) {  
93         revert DebtToken_UnApprovedHolder();  
94     }  
95     if (from == to) {  
96         revert DebtToken_NoSelfTransfer();  
97     }  
98     _spendAllowance(from, _msgSender(), amount);  
99     _transfer(from, to, amount);  
100    return true;  
101 }
```

The code shows that both the `msg.sender` and the `_msgSender()` are being used, as seen in lines 92 and 98. This creates inconsistency in the formatting used for identifying the function caller.

5.11.2. Remediation

Inspex suggests using the same format for identifying the message sender would create consistency in the

code. In this case, it is recommended to change from the `_msgSender()` to the `msg.sender`, for example:

DebtToken.sol

```
87 function transferFrom(  
88     address from,  
89     address to,  
90     uint256 amount  
91 ) public override returns (bool) {  
92     if (!(okHolders[msg.sender] && okHolders[from] && okHolders[to])) {  
93         revert DebtToken_UnApprovedHolder();  
94     }  
95     if (from == to) {  
96         revert DebtToken_NoSelfTransfer();  
97     }  
98     _spendAllowance(from, msg.sender, amount);  
99     _transfer(from, to, amount);  
100     return true;  
101 }
```

5.12. Outdated Compiler Version

ID	IDX-012
Target	MoneyMarketAccountManager MiniFL Rewarder AdminFacet BorrowFacet CollateralFacet DiamondCutFacet DiamondLoupeFacet LendFacet LiquidationFacet NonCollatBorrowFacet OwnershipFacet ViewFacet FixedFeeModel FixedInterestRateModel TripleSlopeModel6 TripleSlopeModel7 LibDiamond LibDoublyLinkedList LibFullMath LibMoneyMarket01 LibReentrancyGuard LibSafeToken LibShareUtil DebtToken InterestBearingToken MoneyMarketDiamond PancakeswapV2IbTokenLiquidationStrategy PancakeswapV2LiquidationStrategy AlpacaV2Oracle
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	Severity: Info Impact: None Likelihood: None
Status	Resolved The Alpaca Finance team has resolved this issue by changing the Solidity version from

0.8.17 to 0.8.19 in commit `48a6bd6491ff730ba5b596cdc073f01fc8bcc53c`.

5.12.1. Description

The Solidity compiler versions specified in the smart contracts were outdated (<https://blog.soliditylang.org/2023/02/22/solidity-0.8.19-release-announcement/>). As the compilers are regularly updated with bug fixes and new features, the latest stable compiler version should be used to compile the smart contracts for best practice.

MoneyMarketAccountManager.sol

```
1 // SPDX-License-Identifier: BUSL
2 pragma solidity 0.8.17;
```

The table below represents the contracts that apply the outdated Solidity compiler version.

File	Version
MoneyMarketAccountManager.sol (L:2)	0.8.17
MiniFL.sol (L:2)	0.8.17
Rewarder.sol (L:2)	0.8.17
AdminFacet.sol (L:2)	0.8.17
BorrowFacet.sol (L:2)	0.8.17
CollateralFacet.sol (L:2)	0.8.17
DiamondCutFacet.sol (L:2)	0.8.17
DiamondLoupeFacet.sol (L:2)	0.8.17
LendFacet.sol (L:2)	0.8.17
LiquidationFacet.sol (L:2)	0.8.17
NonCollatBorrowFacet.sol (L:2)	0.8.17
OwnershipFacet.sol (L:2)	0.8.17
ViewFacet.sol (L:2)	0.8.17
FixedFeeModel.sol (L:2)	0.8.17
FixedInterestRateModel.sol (L:2)	0.8.17
TripleSlopeModel6.sol (L:2)	0.8.17

TripleSlopeModel7.sol (L:2)	0.8.17
LibDiamond.sol (L:2)	0.8.17
LibDoublyLinkedList.sol (L:2)	0.8.17
LibFullMath.sol (L:3)	0.8.17
LibMoneyMarket01.sol (L:2)	0.8.17
LibReentrancyGuard.sol (L:2)	0.8.17
LibSafeToken.sol (L:2)	0.8.17
LibShareUtil.sol (L:2)	0.8.17
DebtToken.sol (L:2)	0.8.17
InterestBearingToken.sol (L:2)	0.8.17
MoneyMarketDiamond.sol (L:2)	0.8.17
PancakeswapV2IbTokenLiquidationStrategy.sol (L:2)	0.8.17
PancakeswapV2LiquidationStrategy.sol (L:2)	0.8.17
AlpacaV2Oracle.sol (L:2)	0.8.17

5.12.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version (<https://github.com/ethereum/solidity/releases>). At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is 0.8.19, for example:

MoneyMarketAccountManager.sol

```

1 // SPDX-License-Identifier: BUSL
2 pragma solidity 0.8.19;
```

6. Appendix

6.1. About Inspex



CYBERSECURITY PROFESSIONAL SERVICE

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement



inspex
CYBERSECURITY PROFESSIONAL SERVICE