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

inspex
CYBERSECURITY PROFESSIONAL SERVICE

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

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
PancakeswapV2IbTokenLiquidationStrategy	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/PancakeswapV2IbTokenLiquidationStrategy.sol
PancakeswapV2LiquidationStrategy	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/5bdfc8c8ce/solidity/contracts/money-market/PancakeswapV2LiquidationStrategy.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

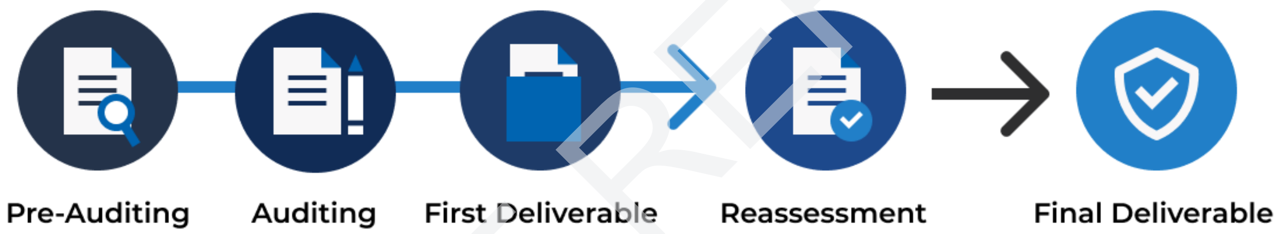
FixedFeeModel	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/fee-models/FixedFeeModel.sol
FixedInterestRateModel	https://github.com/alpaca-finance/alpaca-v2-money-market/blob/48a6bd6491/solidity/contracts/money-market/interest-models/FixedInterestRateModel.sol
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	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. 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**.

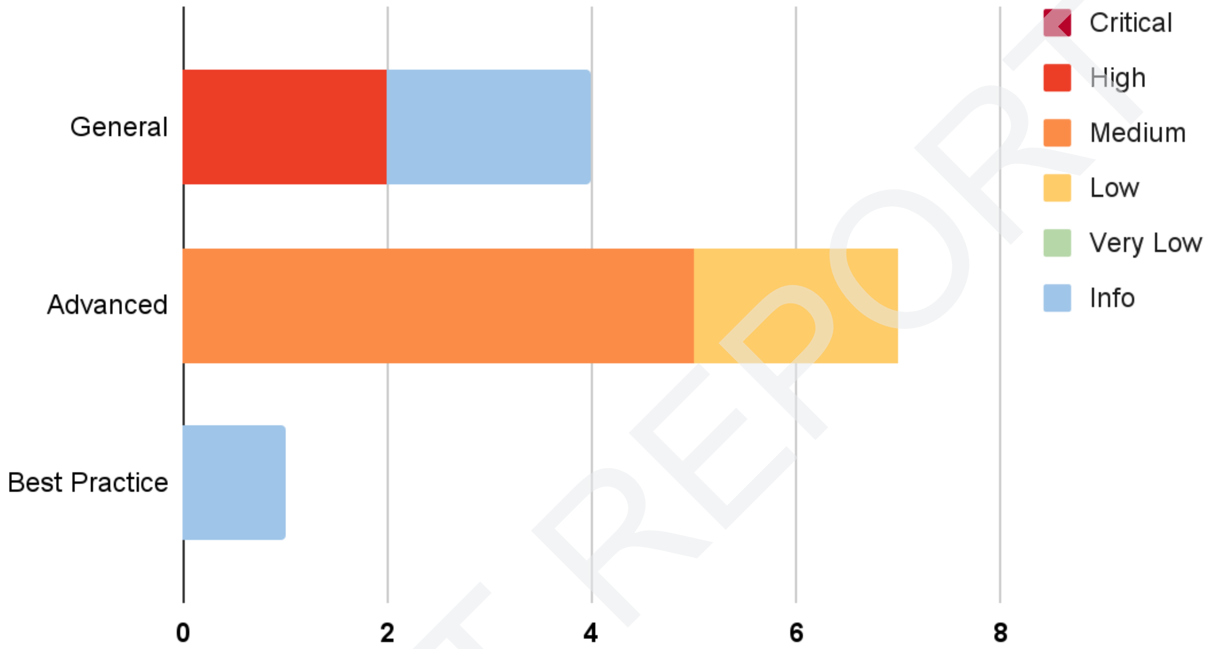
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would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

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

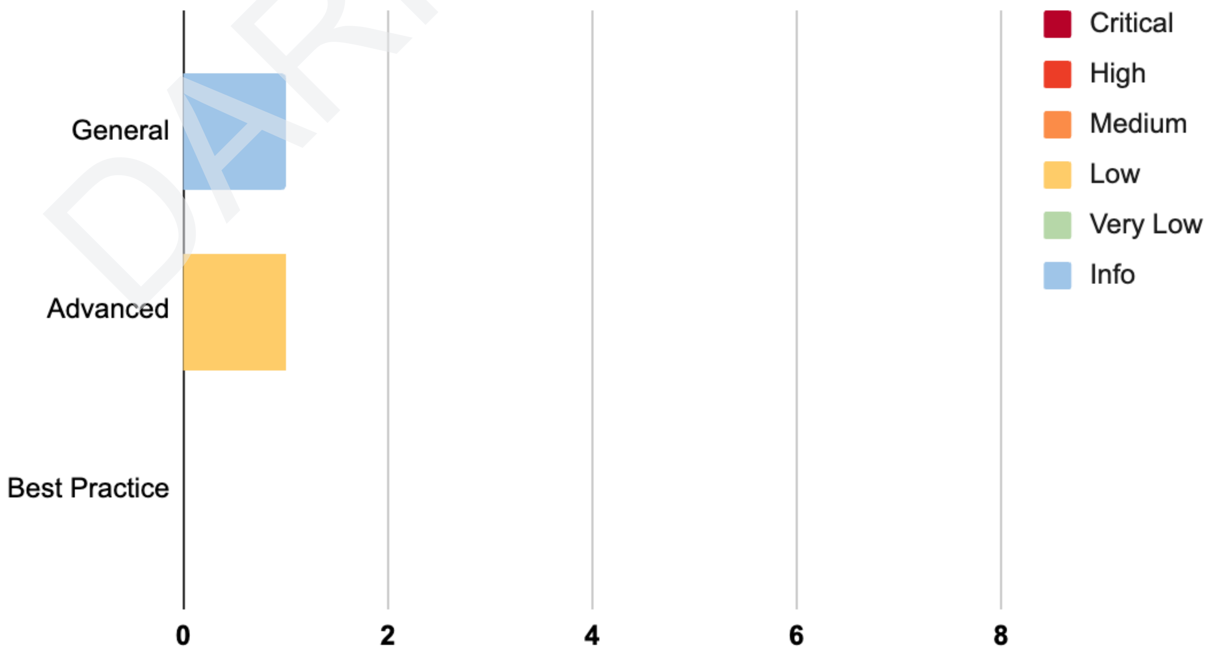
4. Summary of Findings

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	setIbTokenImplementation()	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
Rewarder.sol (L:191)	Rewarder	setRewardPerSecond()	onlyOwner
Rewarder.sol (L:203)	Rewarder	addPool()	onlyOwner
Rewarder.sol (L:228)	Rewarder	setPool()	onlyOwner
Rewarder.sol (L:338)	Rewarder	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 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** This action can be performed by the proxy owner without any restriction.
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	**Severity:** Medium **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 `usedBorrowedPower` of the account. **Likelihood:** Medium When the user decides to repay the borrowed position with full debt amount and the platform has set the `minDebtSize` state to a value greater than zero. Also, the ratio for `moneyMarketDs.overCollatDebtValues` and `moneyMarketDs.overCollatDebtShares` must be indivisible.
Status	Resolved The Alpaca Finance team has resolved this issue by implementing the `shareToValueRoundingUp()` function and applying it to the `repay()` function in commit `48a6bd6491ff730ba5b596cdc073f01fc8bcc53c`.

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
```

```
  IERC20(_token).safeTransferFrom(msg.sender, address(this), _repayAmount);

  // Call repay by forwarding input _debtShareToRepay
  // Money Market should deduct the fund as much as possible
  // If there's excess amount left, transfer back to user
  IERC20(_token).safeApprove(address(moneyMarket), _repayAmount);
  moneyMarket.repay(_account, _subAccountId, _token, _debtShareToRepay);
  // Reset allowance as moneyMarket.repay() might not use all the allowance
  IERC20(_token).safeApprove(address(moneyMarket), 0);

  // Calculate the excess amount left in the contract
  // This will revert if the input repay amount has lower value than _debtShareToRepay
  // And there's some token left in contract (can be done by inject token directly to this contract)
  uint256 _excessAmount = IERC20(_token).balanceOf(address(this)) - _amountBefore;

  if (_excessAmount != 0) {
    IERC20(_token).safeTransfer(msg.sender, _excessAmount);
  }
}
```

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

```
function repay(
  address _account,
  uint256 _subAccountId,
  address _token,
  uint256 _debtShareToRepay
) external nonReentrant {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs = LibMoneyMarket01.moneyMarketDiamondStorage();

  // This function should not be called from anyone
  // except account manager contract and will revert upon trying to do so
  LibMoneyMarket01.onlyAccountManager(moneyMarketDs);

  address _subAccount = LibMoneyMarket01.getSubAccount(_account, _subAccountId);

```

```
    // accrue all debt tokens under subaccount
    // because used borrowing power is calculated from all debt token of sub account
    LibMoneyMarket01.accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);

    // Get the current debt amount and share of this token under the subaccount
    // The current debt share will be used to cap the maximum that can be repaid
    // The current debt amount will be used to check the minimum debt size after repaid
    (uint256 _currentDebtShare, uint256 _currentDebtAmount) = LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
      _subAccount,
      _token,
      moneyMarketDs
    );

    // The debt share that can be repaid should not exceed the current debt share
    // that the subaccount is holding
    uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare, _debtShareToRepay);

    // caching these variables to save gas from multiple reads
    uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
    uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];

    // Find the actual underlying amount that need to be pulled from the share
    uint256 _actualAmountToRepay = LibShareUtil.shareToValue(_actualShareToRepay, _cachedDebtValue, _cachedDebtShare);

    // Pull the token from the account manager, the actual amount received will be used for debt accounting
    // In case somehow there's fee on transfer - which's might be introduced after the token was lent
    // Not reverting to ensure that repay transaction can be done even if there's fee on transfer
    _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender, _actualAmountToRepay);

    // Recalculate the debt share to remove in case there's fee on transfer
    _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay, _cachedDebtShare, _cachedDebtValue);

    // Increase the reserve amount of the token as there's new physical token coming in
    moneyMarketDs.reserves[_token] += _actualAmountToRepay;

    // Check and revert if the repay transaction will violate the business rule
```

```
  // namely the debt size after repaid should be more than minimum debt size
  _validateRepay(
    _token,
    _currentDebtShare,
    _currentDebtAmount,
    _actualShareToRepay,
    _actualAmountToRepay,
    moneyMarketDs
  );

  // Remove the debt share from this subaccount's accounting
  // additionally, this library call will unstake the debt token
  // from miniFL and burn the debt token
  LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
    _account,
    _subAccount,
    _token,
    _actualShareToRepay,
    _actualAmountToRepay,
    moneyMarketDs
  );

  emit LogRepay(_account, _subAccountId, _token, msg.sender,
_actualAmountToRepay);
}
```

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 =
     moneyMarketDs.subAccountDebtShares[_subAccount];
1028   IMiniFL _miniFL = moneyMarketDs.miniFL;
1029
1030   userDebtShare.initIfNotExist();
```

```

    // get share value to update states
    _shareToAdd = LibShareUtil.valueToShareRoundingUp(
      _amount,
      moneyMarketDs.overCollatDebtShares[_token],
      moneyMarketDs.overCollatDebtValues[_token]
    );

    // update over collat debt
    moneyMarketDs.overCollatDebtShares[_token] += _shareToAdd;
    moneyMarketDs.overCollatDebtValues[_token] += _amount;

    // update global debt
    moneyMarketDs.globalDebts[_token] += _amount;

    // update user's debtshare
    userDebtShare.addOrUpdate(_token, userDebtShare.getAmount(_token) +
_shareToAdd);
    // revert if number of debt tokens exceed limit per sub account
    if (userDebtShare.length() > moneyMarketDs.maxNumOfDebtPerSubAccount) {
      revert LibMoneyMarket01_NumberOfTokenExceedLimit();
    }

    // mint debt token to money market and stake to miniFL
    address _debtToken = moneyMarketDs.tokenToDebtTokens[_token];
    uint256 _poolId = moneyMarketDs.miniFLPoolIds[_debtToken];

    // pool for debt token always exist
    // since pool is created during AdminFacet.openMarket()
    IDebtToken(_debtToken).mint(address(this), _shareToAdd);
    IERC20(_debtToken).safeApprove(address(_miniFL), _shareToAdd);
    _miniFL.deposit(_account, _poolId, _shareToAdd);

    emit LogOverCollatBorrow(msg.sender, _subAccount, _token, _amount,
_shareToAdd);
}
```

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

```
function _validateRepay(
  address _repayToken,
  uint256 _currentSubAccountDebtShare,
  uint256 _currentSubAccountDebtAmount,
  uint256 _shareToRepay,
  uint256 _amountToRepay,
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs
) internal view {
  // if partial repay, check if debt after repaid more than minDebtSize
  // no check if repay entire debt
  if (_currentSubAccountDebtShare > _shareToRepay) {
    uint256 _tokenPrice = LibMoneyMarket01.getPriceUSD(_repayToken,
moneyMarketDs);
    // Revert if debt size in USD after repaid is less than minDebtSize
    if (
      ((_currentSubAccountDebtAmount - _amountToRepay) *
        moneyMarketDs.tokenConfigs[_repayToken].to18ConversionFactor *
        _tokenPrice) /
        1e18 <
      moneyMarketDs.minDebtSize
    ) {
      revert BorrowFacet_BorrowLessThanMinDebtSize();
    }
  }
}
```

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;
```

```
    }
    return LibFullMath.mulDiv(_shareAmount, _totalValue, _totalShare);
  }

  function shareToValueRoundingUp(
    uint256 _shareAmount,
    uint256 _totalValue,
    uint256 _totalShare
  ) internal pure returns (uint256) {
    uint256 _values = shareToValue(_shareAmount, _totalValue, _totalShare);
    uint256 _valueShares = valueToShare(_values, _totalShare, _totalValue);
    if (_valueShares + 1 == _shareAmount) {
      _values += 1;
    }
    return _values;
  }

  function valueToShare(
    uint256 _tokenAmount,
    uint256 _totalShare,
    uint256 _totalValue
  ) internal pure returns (uint256) {
    if (_totalShare == 0) {
      return _tokenAmount;
    }
    return LibFullMath.mulDiv(_tokenAmount, _totalShare, _totalValue);
  }

  function valueToShareRoundingUp(
    uint256 _tokenAmount,
    uint256 _totalShare,
    uint256 _totalValue
  ) internal pure returns (uint256) {
    uint256 _shares = valueToShare(_tokenAmount, _totalShare, _totalValue);
    uint256 _shareValues = shareToValue(_shares, _totalValue, _totalShare);
    if (_shareValues + 1 == _tokenAmount) {
      _shares += 1;
    }
    return _shares;
  }
}

```

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

BorrowFacet.sol

```
function repay(
```

```
  address _account,
  uint256 _subAccountId,
  address _token,
  uint256 _debtShareToRepay
) external nonReentrant {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs = LibMoneyMarket01.moneyMarketDiamondStorage();

  // This function should not be called from anyone
  // except account manager contract and will revert upon trying to do so
  LibMoneyMarket01.onlyAccountManager(moneyMarketDs);

  address _subAccount = LibMoneyMarket01.getSubAccount(_account, _subAccountId);

  // accrue all debt tokens under subaccount
  // because used borrowing power is calculated from all debt token of sub account
  LibMoneyMarket01.accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);

  // Get the current debt amount and share of this token under the subaccount
  // The current debt share will be used to cap the maximum that can be repaid
  // The current debt amount will be used to check the minimum debt size after repaid
  (uint256 _currentDebtShare, uint256 _currentDebtAmount) = LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
    _subAccount,
    _token,
    moneyMarketDs
  );

  // The debt share that can be repaid should not exceed the current debt share
  // that the subaccount is holding
  uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare, _debtShareToRepay);

  // caching these variables to save gas from multiple reads
  uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
  uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];

  // Find the actual underlying amount that need to be pulled from the share
  uint256 _actualAmountToRepay = LibShareUtil.shareToValueRoundingUp(_actualShareToRepay, _cachedDebtValue, _cachedDebtShare);

  // Pull the token from the account manager, the actual amount received will be used for debt accounting
```

```
  // In case somehow there's fee on transfer - which's might be introduced after the token was lent
  // Not reverting to ensure that repay transaction can be done even if there's fee on transfer
  _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender, _actualAmountToRepay);

  // Recalculate the debt share to remove in case there's fee on transfer
  _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay, _cachedDebtShare, _cachedDebtValue);

  // Increase the reserve amount of the token as there's new physical token coming in
  moneyMarketDs.reserves[_token] += _actualAmountToRepay;

  // Check and revert if the repay transaction will violate the business rule
  // namely the debt size after repaid should be more than minimum debt size
  _validateRepay(
    _token,
    _currentDebtShare,
    _currentDebtAmount,
    _actualShareToRepay,
    _actualAmountToRepay,
    moneyMarketDs
  );

  // Remove the debt share from this subaccount's accounting
  // additionally, this library call will unstake the debt token
  // from miniFL and burn the debt token
  LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
    _account,
    _subAccount,
    _token,
    _actualShareToRepay,
    _actualAmountToRepay,
    moneyMarketDs
  );

  emit LogRepay(_account, _subAccountId, _token, msg.sender, _actualAmountToRepay);
}
```

5.4. Missing Input Validation in setPoolRewarders

ID	IDX-004
Target	MiniFL
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Medium **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 `_updatePool()` will be triggered, which calls a non-existing pool in the `Rewarder` contract. **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.
Status	Resolved The Alpaca Finance team has resolved this issue by adding input validation in the `setPoolRewarders()` function to ensure that the corresponding pool already exists in commit `48a6bd6491ff730ba5b596cdc073f01fc8bcc53c`.

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

```
function setPoolRewarders(uint256 _pid, address[] calldata _newRewarders)
external onlyOwner {
  uint256 _length = _newRewarders.length;
  // loop to check rewarder should be belong to this MiniFL only
  for (uint256 _i; _i < _length; ) {
    if (IRewarder(_newRewarders[_i]).miniFL() != address(this)) {
      revert MiniFL_BadRewarder();
    }

    unchecked {
      ++_i;
```

```
    }
  }

  rewarders[_pid] = _newRewarders;
}
```

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

```
function _updatePool(uint256 _pid) internal returns (PoolInfo memory) {
  PoolInfo memory _poolInfo = poolInfo[_pid];

  if (_poolInfo.lastRewardTime == 0) revert Rewarder1_PoolNotExisted();

  if (block.timestamp > _poolInfo.lastRewardTime) {
    uint256 _stakedBalance = IMiniFL(miniFL).getStakingReserves(_pid);
    if (_stakedBalance > 0) {
      uint256 _timePast;
      unchecked {
        _timePast = block.timestamp - _poolInfo.lastRewardTime;
      }
      uint256 _rewards = (_timePast * rewardPerSecond * _poolInfo.allocPoint) / totalAllocPoint;

      // increase accRewardPerShare with `_rewards/stakedBalance` amount
      // example:
      //  - oldaccRewardPerShare = 0
      //  - _rewards                = 2000
      //  - stakedBalance               = 10000
      // _poolInfo.accRewardPerShare = oldaccRewardPerShare + (_rewards/stakedBalance)
      // _poolInfo.accRewardPerShare = 0 + 2000/10000 = 0.2
      _poolInfo.accRewardPerShare =
        _poolInfo.accRewardPerShare +
        ((_rewards * ACC_REWARD_PRECISION) / _stakedBalance).toUint128();
    }
    _poolInfo.lastRewardTime = block.timestamp.toUint64();
    poolInfo[_pid] = _poolInfo;
    emit LogUpdatePool(_pid, _poolInfo.lastRewardTime, _stakedBalance, _poolInfo.accRewardPerShare);
  }
  return _poolInfo;
```

```
}
```

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

```
function setPoolRewarders(uint256 _pid, address[] calldata _newRewarders) external onlyOwner {
  require(_pid != 0, "The pool id 0 is not allowed");
  uint256 _length = _newRewarders.length;
  // loop to check rewarder should be belong to this MiniFL only
  for (uint256 _i; _i < _length; ) {
    if (IRewarder(_newRewarders[_i]).miniFL() != address(this)) {
      revert MiniFL_BadRewarder();
    }
    require(IRewarder(_newRewarders[_i]).poolInfo[_pid].lastRewardTime != 0, "The pool has not been created in the rewarder");


    unchecked {
      ++_i;
    }
  }

  rewarders[_pid] = _newRewarders;
}
```

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	**Severity:** Medium **Impact:** High The `accrueNonCollatInterest()` 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. **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.
Status	Resolved The Alpaca Finance team has resolved this issue by implementing a cap for the amount of non-collateral accounts in commit `48a6bd6491ff730ba5b596cdc073f01fc8bcc53c`.

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();
```

```
  uint256 _accountLength = _borrowedAccounts.length;
  address _account;
  uint256 _currentAccountDebt;
  uint256 _accountInterest;
  uint256 _newNonCollatDebtValue;

  // sum up all non collat interest of a token
  for (uint256 _i; _i < _accountLength; ) {
    _account = _borrowedAccounts[_i].token;
    _currentAccountDebt = _borrowedAccounts[_i].amount;

    // calculate interest
    // calculation:
    // _accountInterest = nonCollatInterestRate * _timePast * _currentAccountDebt
    //
    // example:
    //   - nonCollatInterestRate = 0.1
    //   - _timePast             = 3200
    //   - _currentAccountDebt   = 100
    //
    //   _accountInterest        = 0.1 * 3200 * 100
    //                           = 32000
    _accountInterest =
      (getNonCollatInterestRate(_account, _token, moneyMarketDs) * _timePast * _currentAccountDebt) /
      1e18;

    // update non collat debt states
    _newNonCollatDebtValue = _currentAccountDebt + _accountInterest;
    // 1. account debt
    moneyMarketDs.nonCollatAccountDebtValues[_account].addOrUpdate(_token, _newNonCollatDebtValue);
    // 2. token debt
    moneyMarketDs.nonCollatTokenDebtValues[_token].addOrUpdate(_account, _newNonCollatDebtValue);

    // accumulate total non collat interest
    _totalNonCollatInterest += _accountInterest;
    unchecked {
      ++_i;
    }
  }
}
```

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

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

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

AdminFacet.sol

```
function setNonCollatBorrowerOk(address _borrower, bool _isOk) external onlyOwner {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs = LibMoneyMarket01.moneyMarketDiamondStorage();
  if (_isOk == moneyMarketDs.nonCollatBorrowerOk[_borrower]){
    return;
  }
  require(moneyMarketDs.nonCollatCount < 1000, "Non Collateral Account Exceed Capacity");
  moneyMarketDs.nonCollatBorrowerOk[_borrower] = _isOk;
  if (_isOk){
    moneyMarketDs.nonCollatCount += 1;
  } else{
    moneyMarketDs.nonCollatCount -= 1;
  }
  emit LogsetNonCollatBorrowerOk(_borrower, _isOk);
}
```

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	**Severity:** Medium **Impact:** Medium The borrower is paying more interest than expected, as the `writeOffSubAccountsDebt()` 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 `_underlyingToken` due to the token being insufficient. **Likelihood:** Medium It is unlikely that the protocol owner will reset the user debt and will not top up to recover the debt amount.
Status	Resolved 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 `48a6bd6491ff730ba5b596cdc073f01fc8bcc53c`. However, the bad debt will be recovered after the Governance Committee votes to allocate some of protocol's revenue to relieve the lenders' losses.

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;
```

```
  address _subAccount;
  uint256 _shareToRemove;
  uint256 _amountToRemove;

  for (uint256 i; i < _length; ) {
    _token = _inputs[i].token;
    _account = _inputs[i].account;
    _subAccount = LibMoneyMarket01.getSubAccount(_account,
_inputs[i].subAccountId);

    // Revert if the subAccount still have collateral left to be liquidated
    if (moneyMarketDs.subAccountCollats[_subAccount].size != 0) {
      revert AdminFacet_SubAccountHealthy(_subAccount);
    }

    // Accrue interest for token so debt share calculation would be correct
    LibMoneyMarket01.accrueInterest(_token, moneyMarketDs);

    // Get remaining debts of the token under subAccount
    (_shareToRemove, _amountToRemove) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
      _subAccount,
      _token,
      moneyMarketDs
    );

    // Reset debts of the token under subAccount
    LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
      _account,
      _subAccount,
      _token,
      _shareToRemove,
      _amountToRemove,
      moneyMarketDs
    );

    emit LogWriteOffSubAccountDebt(_subAccount, _token, _shareToRemove,
_amountToRemove);

    unchecked {
      ++i;
    }
  }
}
```

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

LibMoneyMarket01.sol

```
function removeOverCollatDebtFromSubAccount(
  address _account,
  address _subAccount,
  address _repayToken,
  uint256 _debtShareToRemove,
  uint256 _debtValueToRemove,
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs
) internal {
  // get current debt share of a token
  uint256 _currentDebtShare =
moneyMarketDs.subAccountDebtShares[_subAccount].getAmount(_repayToken);

  // update states
  //  1. update sub account debt share
  moneyMarketDs.subAccountDebtShares[_subAccount].updateOrRemove(_repayToken,
_currentDebtShare - _debtShareToRemove);
  //  2. update total over collat debt share of a token in money market
  moneyMarketDs.overCollatDebtShares[_repayToken] -= _debtShareToRemove;
  //  3. update total over collat debt value of a token in money market
  moneyMarketDs.overCollatDebtValues[_repayToken] -= _debtValueToRemove;
  //  4. update total debt value of a token in money market
  moneyMarketDs.globalDebts[_repayToken] -= _debtValueToRemove;

  // withdraw debt token from miniFL
  IMiniFL _miniFL = moneyMarketDs.miniFL;
  address _debtToken = moneyMarketDs.tokenToDebtTokens[_repayToken];
  _miniFL.withdraw(_account, moneyMarketDs.miniFLPoolIds[_debtToken],
_debtShareToRemove);

  // burn debt token
  IDebtToken(_debtToken).burn(address(this), _debtShareToRemove);

  emit LogRemoveDebt(_account, _subAccount, _repayToken, _debtShareToRemove,
_debtValueToRemove);
}
```

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

```
function getOverCollatInterestRate(address _token, MoneyMarketDiamondStorage
storage moneyMarketDs)
  internal
  view
```

```
  returns (uint256 _interestRate)
{
  // get interest model of a token
  IInterestRateModel _interestModel = moneyMarketDs.interestModels[_token];
  // return 0 if interest model does not exist
  // otherwise, return interest rate from interest model
  if (address(_interestModel) == address(0)) {
    return 0;
  }
  _interestRate = _interestModel.getInterestRate(
    moneyMarketDs.globalDebts[_token],
    getFloatingBalance(_token, moneyMarketDs)
  );
}

```

TripleSlopeModel6.sol

```
function getInterestRate(uint256 debt, uint256 floating) external pure returns (uint256) {
  if (debt == 0 && floating == 0) return 0;

  uint256 total = debt + floating;
  uint256 utilization = (debt * 100e18) / total;
  if (utilization < CEIL_SLOPE_1) {
    // Less than 85% utilization - 0%-17.5% APY
    return (utilization * MAX_INTEREST_SLOPE_1) / (CEIL_SLOPE_1) / 365 days;
  } else if (utilization < CEIL_SLOPE_2) {
    // Between 85% and 90% - 17.5% APY
    return uint256(MAX_INTEREST_SLOPE_2) / 365 days;
  } else if (utilization < CEIL_SLOPE_3) {
    // Between 90% and 100% - 17.5%-150% APY
    return
      (MAX_INTEREST_SLOPE_2 +
        ((utilization - CEIL_SLOPE_2) * (MAX_INTEREST_SLOPE_3 - MAX_INTEREST_SLOPE_2)) /
        (CEIL_SLOPE_3 - CEIL_SLOPE_2)) / 365 days;
  } else {
    // Not possible, but just in case - 150% APY
    return MAX_INTEREST_SLOPE_3 / 365 days;
  }
}
```

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*100/100=90	80*100/100=80	80*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

```
function topUpTokenReserve(address _token, uint256 _amount) external onlyOwner
{
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
LibMoneyMarket01.moneyMarketDiamondStorage();

  // Prevent topup token that didn't have market
  if (moneyMarketDs.tokenToIbTokens[_token] == address(0)) revert
AdminFacet_InvalidToken(_token);

  // Allow topup for token that has fee on transfer
  uint256 _actualAmountReceived = LibMoneyMarket01.unsafePullTokens(_token,
msg.sender, _amount);
  moneyMarketDs.reserves[_token] += _actualAmountReceived;

  emit LogTopUpTokenReserve(_token, _actualAmountReceived);
}
```


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

```
function topUpTokenReserve(address _token, uint256 _amount) internal {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
LibMoneyMarket01.moneyMarketDiamondStorage();

  // Prevent topup token that didn't have market
  if (moneyMarketDs.tokenToIbTokens[_token] == address(0)) revert
AdminFacet_InvalidToken(_token);

```

```
  // Allow topup for token that has fee on transfer
  uint256 _actualAmountReceived = LibMoneyMarket01.unsafePullTokens(_token, msg.sender, _amount);
  moneyMarketDs.reserves[_token] += _actualAmountReceived;

  emit LogTopUpTokenReserve(_token, _actualAmountReceived);
}
```

AdminFacet.sol

```
function writeOffSubAccountsDebt(WriteOffSubAccountDebtInput[] calldata _inputs) external onlyOwner {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs = LibMoneyMarket01.moneyMarketDiamondStorage();

  uint256 _length = _inputs.length;

  address _token;
  address _account;
  address _subAccount;
  uint256 _shareToRemove;
  uint256 _amountToRemove;

  for (uint256 i; i < _length; ) {
    _token = _inputs[i].token;
    _account = _inputs[i].account;
    _subAccount = LibMoneyMarket01.getSubAccount(_account, _inputs[i].subAccountId);

    // Revert if the subAccount still have collateral left to be liquidated
    if (moneyMarketDs.subAccountCollats[_subAccount].size != 0) {
      revert AdminFacet_SubAccountHealthy(_subAccount);
    }

    // Accrue interest for token so debt share calculation would be correct
    LibMoneyMarket01.accrueInterest(_token, moneyMarketDs);

    // Get remaining debts of the token under subAccount
    (_shareToRemove, _amountToRemove) = LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
      _subAccount,
      _token,
      moneyMarketDs
    );

    // Reset debts of the token under subAccount
    LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
      _account,
```

```
      _subAccount,
      _token,
      _shareToRemove,
      _amountToRemove,
      moneyMarketDs
    );

    topUpTokenReserve(_token, _amountToRemove);

    emit LogWriteOffSubAccountDebt(_subAccount, _token, _shareToRemove,
_amountToRemove);

    unchecked {
      ++i;
    }
  }
}
```

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	**Severity:** Medium **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 `globalDebt` state value and the borrower loses 1 Wei of the collateral, resulting in extra interest for the lenders and the borrowers. **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 `debtValue`.
Status	Resolved The Alpaca Finance team has resolved this issue by implementing the rounding up condition for the `repayAmountWithFee` when the user repurchasing the entire position in commit `48a6bd6491ff730ba5b596cdc073f01fc8bcc53c`.

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      );
```

```
    uint256 _maxAmountRepurchaseable = (_currentDebtAmount *
      (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS)) / LibMoneyMarket01.MAX_BPS;

    if (_desiredRepayAmount > _maxAmountRepurchaseable) {
      _vars.repayAmountWithFee = _maxAmountRepurchaseable;
      _vars.repayAmountWithoutFee = _currentDebtAmount;
    } else {
      _vars.repayAmountWithFee = _desiredRepayAmount;
      _vars.repayAmountWithoutFee =
        (_desiredRepayAmount * LibMoneyMarket01.MAX_BPS) /
        (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS);
    }

    _vars.repurchaseFeeToProtocol = _vars.repayAmountWithFee - _vars.repayAmountWithoutFee;
  }
```

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,
```

```
        _collatToken,
        _collatAmountOut,
        moneyMarketDs
    );
```

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

LiquidationFacet.sol

```
function removeOverCollatDebtFromSubAccount(
    address _account,
    address _subAccount,
    address _repayToken,
    uint256 _debtShareToRemove,
    uint256 _debtValueToRemove,
    LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs
) internal {
    // get current debt share of a token
    uint256 _currentDebtShare =
moneyMarketDs.subAccountDebtShares[_subAccount].getAmount(_repayToken);

    // update states
    //  1. update sub account debt share
    moneyMarketDs.subAccountDebtShares[_subAccount].updateOrRemove(_repayToken,
_currentDebtShare - _debtShareToRemove);
    //  2. update total over collat debt share of a token in money market
    moneyMarketDs.overCollatDebtShares[_repayToken] -= _debtShareToRemove;
    //  3. update total over collat debt value of a token in money market
    moneyMarketDs.overCollatDebtValues[_repayToken] -= _debtValueToRemove;
    //  4. update total debt value of a token in money market
    moneyMarketDs.globalDebts[_repayToken] -= _debtValueToRemove;

    // withdraw debt token from miniFL
    IMiniFL _miniFL = moneyMarketDs.miniFL;
    address _debtToken = moneyMarketDs.tokenToDebtTokens[_repayToken];
    _miniFL.withdraw(_account, moneyMarketDs.miniFLPoolIds[_debtToken],
_debtShareToRemove);

    // burn debt token
    IDebtToken(_debtToken).burn(address(this), _debtShareToRemove);

    emit LogRemoveDebt(_account, _subAccount, _repayToken, _debtShareToRemove,
_debtValueToRemove);
}
```

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

```
struct RepurchaseLocalVars {
  address subAccount;
  uint256 totalBorrowingPower;
  uint256 usedBorrowingPower;
  uint256 repayAmountWithFee;
  uint256 repurchaseFeeToProtocol;
  uint256 repurchaseRewardBps;
  uint256 repayAmountWithoutFee;
  uint256 repayTokenPrice;
  bool isPurchaseAll;
}
```

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

LiquidationFacet.sol

```
  {
    (, uint256 _currentDebtAmount) =
LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
      _vars.subAccount,
      _repayToken,
      moneyMarketDs
    );
    uint256 _maxAmountRepurchaseable = (_currentDebtAmount *
      (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS)) /
LibMoneyMarket01.MAX_BPS;

    if (_desiredRepayAmount > _maxAmountRepurchaseable) {
      _vars.repayAmountWithFee = _maxAmountRepurchaseable;
      _vars.repayAmountWithoutFee = _currentDebtAmount;
      _vars.isPurchaseAll = true;
    } else {
      _vars.repayAmountWithFee = _desiredRepayAmount;
      _vars.repayAmountWithoutFee =
        (_desiredRepayAmount * LibMoneyMarket01.MAX_BPS) /
        (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS);
      _vars.isPurchaseAll = false;
    }

    _vars.repurchaseFeeToProtocol = _vars.repayAmountWithFee -
```

```
_vars.repayAmountWithoutFee;
  }
```

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

LiquidationFacet.sol

```
uint256 _actualRepayAmountWithoutFee;
{
  if (_vars.isPurchaseAll){
    uint256 _repayAmountWithoutFee = _vars.repayAmountWithFee -
_vars.repurchaseFeeToProtocol;
    uint256 _actualRepayAmountAsShare = LibShareUtil.valueToShare(
      _repayAmountWithoutFee,
      moneyMarketDs.overCollatDebtShares[_repayToken],
      moneyMarketDs.overCollatDebtValues[_repayToken]
    );
    uint256 _expectRepayAmountAsShare = LibShareUtil.valueToShareRoundingUp(
    _repayAmountWithoutFee,
    moneyMarketDs.overCollatDebtShares[_repayToken],
    moneyMarketDs.overCollatDebtValues[_repayToken]
    );
    if (_actualRepayAmountAsShare + 1 == _expectRepayAmountAsShare) {
      _vars.repayAmountWithFee += 1;
    }
  }
  _actualRepayAmountWithoutFee = LibMoneyMarket01.unsafePullTokens(
    _repayToken,
    msg.sender,
    _vars.repayAmountWithFee
  ) - _vars.repurchaseFeeToProtocol;
}
```

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	**Severity:** Low **Impact:** Medium 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. **Likelihood:** Low 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 `deposit()` function (applying the `pullExactTokens()` function). Hence, it is unlikely that the platform will allow the deflationary token to be used.
Status	Resolved The Alpaca Finance team has clarified that they will not support the deflationary token.

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

```
/// @dev SafeTransferFrom that return actual amount received
/// @param _token The token address
/// @param _from The address to pull token from
/// @param _amount The amount to pull
/// @return _actualAmountReceived The actual amount received
function unsafePullTokens(
  address _token,
  address _from,
```

```
  uint256 _amount
) internal returns (uint256 _actualAmountReceived) {
  // get the token balance of money market before transfer
  uint256 _balanceBefore = IERC20(_token).balanceOf(address(this));
  // transfer token from _from to money market
  IERC20(_token).safeTransferFrom(_from, address(this), _amount);
  // return actual amount received = balance after transfer - balance before transfer
  _actualAmountReceived = IERC20(_token).balanceOf(address(this)) - _balanceBefore;
}
```

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

LibMoneyMarket01.sol

```
/// @dev SafeTransferFrom that revert when not receiving full amount (have fee on transfer)
/// @param _token The token address
/// @param _from The address to pull token from
/// @param _amount The amount to pull
function pullExactTokens(
  address _token,
  address _from,
  uint256 _amount
) internal {
  // get the token balance of money market before transfer
  uint256 _balanceBefore = IERC20(_token).balanceOf(address(this));
  // transfer token from _from to money market
  IERC20(_token).safeTransferFrom(_from, address(this), _amount);
  // check if the token balance of money market is increased by _amount
  // if fee on transer tokens is not supported this will revert
  if (IERC20(_token).balanceOf(address(this)) - _balanceBefore != _amount) {
    revert LibMoneyMarket01_FeeOnTransferTokensNotSupported();
  }
}
```


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

```
function repurchase(
  address _account,
  uint256 _subAccountId,
  address _repayToken,
  address _collatToken,
  uint256 _desiredRepayAmount
) external nonReentrant returns (uint256 _collatAmountOut) {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs = LibMoneyMarket01.moneyMarketDiamondStorage();

  // We only allow EOA or whitelisted contract to repurchase
  // Revert if caller is contract that is not whitelisted
  // `msg.sender != tx.origin` means that `msg.sender` is contract
  if (msg.sender != tx.origin && !moneyMarketDs.repurchasersOk[msg.sender]) {
    revert LiquidationFacet_Unauthorized();
  }

  RepurchaseLocalVars memory _vars;

  _vars.subAccount = LibMoneyMarket01.getSubAccount(_account, _subAccountId);

  // Accrue all debt tokens under subaccount
  // Because used borrowing power is calculated from all debt token of the subaccount
  LibMoneyMarket01.accrueBorrowedPositionsOf(_vars.subAccount, moneyMarketDs);

  _vars.totalBorrowingPower = LibMoneyMarket01.getTotalBorrowingPower(_vars.subAccount, moneyMarketDs);
  (_vars.usedBorrowingPower, ) = LibMoneyMarket01.getTotalUsedBorrowingPower(_vars.subAccount, moneyMarketDs);
  // Revert if position is not repurchasable (borrowingPower / usedBorrowingPower >= 1)
  if (_vars.totalBorrowingPower >= _vars.usedBorrowingPower) {
    revert LiquidationFacet_Healthy();
  }

  // Cap repurchase amount if needed and calculate fee
  // Fee is calculated from repaid amount
  //
  // formulas:
  //     maxAmountRepurchaseable  = currentDebt * (1 + fee)
  //     repurchaseFee            = repayAmountWithFee - repayAmountWithoutFee
  //     if desiredRepayAmount > maxAmountRepurchaseable
  //        repayAmountWithFee    = maxAmountRepurchaseable
  //        repayAmountWithoutFee = currentDebt
  //     else
  //        repayAmountWithFee    = desiredRepayAmount
```

```
//        repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
//
// calculation example:
//     assume 1 eth = 2000 USD, 10% repurchase fee, ignore collat,borrowingFactor, no premium
//     collateral: 2000 USDC
//     debt       : 1 eth
//     maxAmountRepurchaseable = currentDebt * (1 + fee)
//                             = 1 * (1 + 0.1) = 1.1 eth
//
//     ex 1: desiredRepayAmount > maxAmountRepurchaseable
//     input : desiredRepayAmount = 1.2 eth
//     repayAmountWithFee     = maxAmountRepurchaseable = 1.1 eth
//     repayAmountWithoutFee = currentDebt = 1 eth
//     repurchaseFee          = repayAmountWithFee - repayAmountWithoutFee
//                            = 1.1 - 1 = 0.1 eth
//
//     ex 2: desiredRepayAmount < currentDebt
//     input : desiredRepayAmount = 0.9 eth
//     repayAmountWithFee     = desiredRepayAmount = 0.9 eth
//     repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
//                            = 0.9 / (1 + 0.1) = 0.818181.. eth
//     repurchaseFee          = repayAmountWithFee - repayAmountWithoutFee
//                            = 0.9 - 0.818181... = 0.0818181... eth
//
//     ex 3: desiredRepayAmount == currentDebt
//     input : desiredRepayAmount = 1 eth
//     repayAmountWithFee     = desiredRepayAmount = 1 eth
//     repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
//                            = 1 / (1 + 0.1) = 0.909090.. eth
//     repurchaseFee          = repayAmountWithFee - repayAmountWithoutFee
//                            = 1 - 0.909090... = 0.0909090... eth
//
//     ex 4: currentDebt < desiredRepayAmount < maxAmountRepurchaseable
//     input : desiredRepayAmount = 1.05 eth
//     repayAmountWithFee = desiredRepayAmount = 1.05 eth
//     repayAmountWithoutFee = desiredRepayAmount / (1 + fee)
//                            = 1.05 / (1 + 0.1) = 0.9545454... eth
//     repurchaseFee          = repayAmountWithFee - repayAmountWithoutFee
//                            = 1.05 - 0.9545454... = 0.09545454... eth
//
{
  (, uint256 _currentDebtAmount) = LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
      _vars.subAccount,
      _repayToken,
      moneyMarketDs
```

```
    );
    uint256 _maxAmountRepurchaseable = (_currentDebtAmount *
      (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS)) / LibMoneyMarket01.MAX_BPS;

    if (_desiredRepayAmount > _maxAmountRepurchaseable) {
      _vars.repayAmountWithFee = _maxAmountRepurchaseable;
      _vars.repayAmountWithoutFee = _currentDebtAmount;
    } else {
      _vars.repayAmountWithFee = _desiredRepayAmount;
      _vars.repayAmountWithoutFee =
        (_desiredRepayAmount * LibMoneyMarket01.MAX_BPS) /
        (moneyMarketDs.repurchaseFeeBps + LibMoneyMarket01.MAX_BPS);
    }

    _vars.repurchaseFeeToProtocol = _vars.repayAmountWithFee - _vars.repayAmountWithoutFee;
  }

  _vars.repayTokenPrice = LibMoneyMarket01.getPriceUSD(_repayToken, moneyMarketDs);

  // Revert if repayment exceeds threshold (repayment > maxLiquidateThreshold * usedBorrowingPower)
  _validateBorrowingPower(_repayToken, _vars.repayAmountWithoutFee, _vars.usedBorrowingPower, moneyMarketDs);

  // Get dynamic repurchase reward to further incentivize repurchase
  _vars.repurchaseRewardBps = moneyMarketDs.repurchaseRewardModel.getFeeBps(
    _vars.totalBorrowingPower,
    _vars.usedBorrowingPower
  );

  // Calculate payout for repurchaser (collateral with premium)
  {
    uint256 _collatTokenPrice = LibMoneyMarket01.getPriceUSD(_collatToken, moneyMarketDs);

    uint256 _repayTokenPriceWithPremium = (_vars.repayTokenPrice *
      (LibMoneyMarket01.MAX_BPS + _vars.repurchaseRewardBps)) / LibMoneyMarket01.MAX_BPS;

    // collatAmountOut = repayAmount * repayTokenPriceWithPremium / collatTokenPrice
    _collatAmountOut =
      (_vars.repayAmountWithFee *
        _repayTokenPriceWithPremium *
```

```
        moneyMarketDs.tokenConfigs[_repayToken].to18ConversionFactor) /
      (_collatTokenPrice * moneyMarketDs.tokenConfigs[_collatToken].to18ConversionFactor);

    // revert if subAccount collat is not enough to cover desired repay amount
    // this could happen when there are multiple small collat and one large debt
    // ex. assume 1 eth = 2000 USD, no repurchase fee or premium, ignore collat,borrowingFactor
    //     collateral : 1000 USDT, 1000 USDC
    //     debt       : 1 eth
    //     input      : desiredRepayAmount = 0.6 eth, collatToken = USDC
    //     collatAmountOut = repayAmount * repayTokenPrice / collatTokenPrice
    //                     = 0.6 * 2000 / 1 = 1200 USDC
    //     this should revert since there is not enough USDC collateral to be repurchased
    if (_collatAmountOut > moneyMarketDs.subAccountCollats[_vars.subAccount].getAmount(_collatToken)) {
      revert LiquidationFacet_InsufficientAmount();
    }
  }

  ////////////////////////////////
  // EFFECTS & INTERACTIONS
  ////////////////////////////////

  // Transfer repay token in
  // In case of token with fee on transfer, debt would be repaid by amount after transfer fee
  // which won't be able to repurchase entire position
  // repaidAmount = amountReceived - repurchaseFee
  uint256 _actualRepayAmountWithoutFee = LibMoneyMarket01.unsafePullTokens(
    _repayToken,
    msg.sender,
    _vars.repayAmountWithFee
  ) - _vars.repurchaseFeeToProtocol;

  // Remove subAccount debt
  LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
    _account,
    _vars.subAccount,
    _repayToken,
    LibShareUtil.valueToShare(
      _actualRepayAmountWithoutFee,
      moneyMarketDs.overCollatDebtShares[_repayToken],
      moneyMarketDs.overCollatDebtValues[_repayToken]
    ),
```

```
    _actualRepayAmountWithoutFee,
    moneyMarketDs
  );
  // need to call removeCollat which might withdraw from miniFL to be able to transfer to repurchaser
  LibMoneyMarket01.removeCollatFromSubAccount(
    _account,
    _vars.subAccount,
    _collatToken,
    _collatAmountOut,
    moneyMarketDs
  );

  // Increase reserves balance with repaid tokens
  // Safe to use unchecked because _actualRepayAmountWithoutFee is derived from balanceOf
  unchecked {
    moneyMarketDs.reserves[_repayToken] += _actualRepayAmountWithoutFee;
  }

  // Transfer collat token with premium back to repurchaser
  IERC20(_collatToken).safeTransfer(msg.sender, _collatAmountOut);
  // Transfer protocol's repurchase fee to treasury
  IERC20(_repayToken).safeTransfer(moneyMarketDs.liquidationTreasury, _vars.repurchaseFeeToProtocol);

  emit LogRepurchase(
    msg.sender,
    _account,
    _subAccountId,
    _repayToken,
    _collatToken,
    _actualRepayAmountWithoutFee,
    _collatAmountOut,
    _vars.repurchaseFeeToProtocol,
    (_collatAmountOut * _vars.repurchaseRewardBps) / LibMoneyMarket01.MAX_BPS
  );
}
```

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	**Severity:** Low **Impact:** Medium The interest accrued by the Alpaca protocol increases with every call of the `accrueInterest()` function. Therefore, a borrower and lender will pay or receive less interest if the function is called fewer times. **Likelihood:** Low This issue is unlikely to occur, since the platform implements the external `accrueInterest()` function in the `BorrowFacet` facet that enables the bot to call periodically. Moreover, the `accrueInterest()` is called internally for all functionalities that are related to the interest flow.
Status	**Acknowledged** The Alpaca Finance team has acknowledged this issue.

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

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

```
    unchecked {
      _secondsSinceLastAccrual = block.timestamp - _lastAccrualTimestamp;
    }
    // accrue interest
    uint256 _overCollatInterest = accrueOverCollatInterest(_token, _secondsSinceLastAccrual, moneyMarketDs);
    uint256 _nonCollatInterest = accrueNonCollatInterest(_token, _secondsSinceLastAccrual, moneyMarketDs);

    // update global debt
    uint256 _totalInterest = _overCollatInterest + _nonCollatInterest;
    moneyMarketDs.globalDebts[_token] += _totalInterest;

    // update timestamp
    moneyMarketDs.debtLastAccruedAt[_token] = block.timestamp;

    // book protocol's revenue
    // calculation:
    // _protocolFee = (_totalInterest * lendingFeeBps) / MAX_BPS
    //
    // example:
    //   - _totalInterest = 1
    //   - lendingFeeBps  = 1900
    //   - MAX_BPS        = 10000
    //
    //   _protocolFee     = (1 * 1900) / 10000
    //                    = 0.19
    uint256 _protocolFee = (_totalInterest * moneyMarketDs.lendingFeeBps) / MAX_BPS;
    moneyMarketDs.protocolReserves[_token] += _protocolFee;

    emit LogAccrueInterest(_token, _totalInterest, _protocolFee);
  }
}
```


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 `AccountManager` 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

```
function repay(
  address _account,
  uint256 _subAccountId,
  address _token,
  uint256 _debtShareToRepay
) external nonReentrant {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
LibMoneyMarket01.moneyMarketDiamondStorage();

  // This function should not be called from anyone
  // except account manager contract and will revert upon trying to do so
  LibMoneyMarket01.onlyAccountManager(moneyMarketDs);

  address _subAccount = LibMoneyMarket01.getSubAccount(_account,
_subAccountId);

  // accrue all debt tokens under subaccount
```

```
  // because used borrowing power is calculated from all debt token of sub account
  LibMoneyMarket01.accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);

  // Get the current debt amount and share of this token under the subaccount
  // The current debt share will be used to cap the maximum that can be repaid
  // The current debt amount will be used to check the minimum debt size after repaid
  (uint256 _currentDebtShare, uint256 _currentDebtAmount) = LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
    _subAccount,
    _token,
    moneyMarketDs
  );

  // The debt share that can be repaid should not exceed the current debt share
  // that the subaccount is holding
  uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare, _debtShareToRepay);

  // caching these variables to save gas from multiple reads
  uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
  uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];

  // Find the actual underlying amount that need to be pulled from the share
  uint256 _actualAmountToRepay = LibShareUtil.shareToValue(_actualShareToRepay, _cachedDebtValue, _cachedDebtShare);

  // Pull the token from the account manager, the actual amount received will be used for debt accounting
  // In case somehow there's fee on transfer - which's might be introduced after the token was lent
  // Not reverting to ensure that repay transaction can be done even if there's fee on transfer
  _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender, _actualAmountToRepay);

  // Recalculate the debt share to remove in case there's fee on transfer
  _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay, _cachedDebtShare, _cachedDebtValue);

  // Increase the reserve amount of the token as there's new physical token coming in
  moneyMarketDs.reserves[_token] += _actualAmountToRepay;

  // Check and revert if the repay transaction will violate the business rule
  // 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   }
```

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 `_caller` parameter should be the address of the account that called the function in the `MoneyMarketAccountManager` contract
AdminFacet.sol(L:57)	LogWriteOffSubAccountDebt()	subAccount	The `account` parameter should be logged along with the `subAccount` parameter
NonCollatBorrowFacet.sol(L:21)	LogNonCollatBorrow()	_removeDebtAmount	The parameter's name should be declared according to its value,

			e.g., `_borrowAmount`
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()	_caller	The `_caller` parameter should be the address of the account that called the function in the `MoneyMarketAccountManager` contract
LibMoneyMarket01.sol(L:1063)	LogOverCollatBorrow()	_account	The `_account` variable should be used as the `_account` parameter instead of the `msg.sender` 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

```
function repay(
  address _account,
  uint256 _subAccountId,
  address _token,
  uint256 _debtShareToRepay,
  address _caller
) external nonReentrant {
  LibMoneyMarket01.MoneyMarketDiamondStorage storage moneyMarketDs =
LibMoneyMarket01.moneyMarketDiamondStorage();

  // This function should not be called from anyone
  // except account manager contract and will revert upon trying to do so
  LibMoneyMarket01.onlyAccountManager(moneyMarketDs);

  address _subAccount = LibMoneyMarket01.getSubAccount(_account,
_subAccountId);

```

```
  // accrue all debt tokens under subaccount
  // because used borrowing power is calculated from all debt token of sub account
  LibMoneyMarket01.accrueBorrowedPositionsOf(_subAccount, moneyMarketDs);

  // Get the current debt amount and share of this token under the subaccount
  // The current debt share will be used to cap the maximum that can be repaid
  // The current debt amount will be used to check the minimum debt size after repaid
  (uint256 _currentDebtShare, uint256 _currentDebtAmount) = LibMoneyMarket01.getOverCollatDebtShareAndAmountOf(
    _subAccount,
    _token,
    moneyMarketDs
  );

  // The debt share that can be repaid should not exceed the current debt share
  // that the subaccount is holding
  uint256 _actualShareToRepay = LibFullMath.min(_currentDebtShare, _debtShareToRepay);

  // caching these variables to save gas from multiple reads
  uint256 _cachedDebtValue = moneyMarketDs.overCollatDebtValues[_token];
  uint256 _cachedDebtShare = moneyMarketDs.overCollatDebtShares[_token];

  // Find the actual underlying amount that need to be pulled from the share
  uint256 _actualAmountToRepay = LibShareUtil.shareToValue(_actualShareToRepay, _cachedDebtValue, _cachedDebtShare);

  // Pull the token from the account manager, the actual amount received will be used for debt accounting
  // In case somehow there's fee on transfer - which's might be introduced after the token was lent
  // Not reverting to ensure that repay transaction can be done even if there's fee on transfer
  _actualAmountToRepay = LibMoneyMarket01.unsafePullTokens(_token, msg.sender, _actualAmountToRepay);

  // Recalculate the debt share to remove in case there's fee on transfer
  _actualShareToRepay = LibShareUtil.valueToShare(_actualAmountToRepay, _cachedDebtShare, _cachedDebtValue);

  // Increase the reserve amount of the token as there's new physical token coming in
  moneyMarketDs.reserves[_token] += _actualAmountToRepay;

  // Check and revert if the repay transaction will violate the business rule
```

```
    // namely the debt size after repaid should be more than minimum debt size
    _validateRepay(
      _token,
      _currentDebtShare,
      _currentDebtAmount,
      _actualShareToRepay,
      _actualAmountToRepay,
      moneyMarketDs
    );

    // Remove the debt share from this subaccount's accounting
    // additionally, this library call will unstake the debt token
    // from miniFL and burn the debt token
    LibMoneyMarket01.removeOverCollatDebtFromSubAccount(
      _account,
      _subAccount,
      _token,
      _actualShareToRepay,
      _actualAmountToRepay,
      moneyMarketDs
    );

    emit LogRepay(_account, _subAccountId, _token, _caller, _actualAmountToRepay);
  }
```

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

MoneyMarketAccountManager.sol

```
function repayFor(
  address _account,
  uint256 _subAccountId,
  address _token,
  uint256 _repayAmount,
  uint256 _debtShareToRepay
) external {
  // cache the balance of token before proceeding
  uint256 _amountBefore = IERC20(_token).balanceOf(address(this));

  // Fund this contract from caller
  // ignore the fact that there might be fee on transfer
  IERC20(_token).safeTransferFrom(msg.sender, address(this), _repayAmount);

  // Call repay by forwarding input _debtShareToRepay
  // Money Market should deduct the fund as much as possible
  // If there's excess amount left, transfer back to user
  IERC20(_token).safeApprove(address(moneyMarket), _repayAmount);
```

```
  moneyMarket.repay(_account, _subAccountId, _token, _debtShareToRepay, msg.sender);
  // Reset allowance as moneyMarket.repay() might not use all the allowance
  IERC20(_token).safeApprove(address(moneyMarket), 0);

  // Calculate the excess amount left in the contract
  // This will revert if the input repay amount has lower value than _debtShareToRepay
  // And there's some token left in contract (can be done by inject token directly to this contract)
  uint256 _excessAmount = IERC20(_token).balanceOf(address(this)) - _amountBefore;

  if (_excessAmount != 0) {
    IERC20(_token).safeTransfer(msg.sender, _excessAmount);
  }
}
```

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 `msg.sender` instead of `_msgSender()` in commit `48a6bd6491ff730ba5b596cdc073f01fc8bcc53c`.

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

```
function transferFrom(
  address from,
  address to,
  uint256 amount
) public override returns (bool) {
  if (!(okHolders[msg.sender] && okHolders[from] && okHolders[to])) {
    revert DebtToken_UnApprovedHolder();
  }
  if (from == to) {
    revert DebtToken_NoSelfTransfer();
  }
  _spendAllowance(from, msg.sender, amount);
  _transfer(from, to, amount);
  return true;
}
```

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

```
// SPDX-License-Identifier: BUSL
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

```
// SPDX-License-Identifier: BUSL
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