BAIL security

Lista Dao
Credit Loan

FINAL REPORT

January ‘2026

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	ListaDao - Credit Loan - Audit Report
Website	lista.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/lista-dao/moolah/tree/cca375aead835fe9ea1461c47fd26b1ed7242728
Resolution 1	https://github.com/lista-dao/moolah/tree/6ea7e37a9be2fe4a732663a6cb2957976aaa924c

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no changes made)	Failed resolution	Open
High	9	5		4		
Medium	18	3		15		
Low	17	6		7		4
Informational	15	5		7		3
Governance	1					1
Total	60	19		33		8

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Global

Issue_01	Governance Issue
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. Governance can upgrade all contracts and has various different mechanisms to set roles that can execute malicious actions.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	

Issue_02	UUPSUpgradeable/PausableUpgradeable is not initialized
Severity	**Informational**
Description	Currently, the UUPSUpgradeable inheritance contract is not initialized. While this does not expose any harm, it is considered as best practice. It is applicable to all contracts and likewise Pausable is not initialized.
Recommendations	Consider initializing the UUPSUpgradeable/PausableUpgradeable contracts.
Comments / Resolution	Fixed. __Pausable_init_unchained is now invoked during initialization of CreditToken

Issue_03	Lack of support for transfer-tax tokens
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using such tokens.
Comments / Resolution	

Broker

CreditBroker

The CreditBroker contract is an upgradeable fixed-term loan broker that lets users borrow a configured loan token from a Moolah market while posting a credit token as collateral, with the broker maintaining a separate fixed-position debt ledger (principal + interest + potential late penalties) on top of Moolah's share-based borrow accounting.

The scope boundary is explicit in how responsibilities are split:

- Moolah is the custody and accounting layer for the *canonical* market position (collateral balance and borrow shares) and is the place where borrowing/repaying collateralized debt ultimately occurs.
- CreditBroker is the policy and bookkeeping layer that:
    - defines the available fixed-term products (fixedTerms)
    - records each user's fixed loan positions (fixedLoanPositions)
    - computes interest/penalties via CreditBrokerMath
    - and routes "revenue" (interest and penalties) to a relayer for eventual supply into the Moolah vault
- CreditBrokerInterestRelayer is used as an operational buffer: the broker pushes interest/penalty amounts to the relayer, and the relayer supplies into Moolah once the accumulated amount exceeds Moolah's minLoan.

Besides standard supplyCollateral/withdrawCollateral/borrow/repay executions, the CreditBroker supports a repayment with LISTA tokens which then allows users to repay at a certain discount.

Appendix: Fixed Term Loans

Fixed-term loans require the broker to track repayment state independently from Moolah borrow shares so it can compute interest and penalties in a deterministic way.

Accounting rules implemented by the broker:

- Each borrow creates a FixedLoanPosition with:
  - principal fixed at borrow time
  - principalRepaid and interestRepaid incremented over time
  - a [start, end) term window
  - a termType selecting which interest model applies
  - and a noInterestUntil window for upfront-interest positions

- On repayment:
  - interest is paid first and booked into interestRepaid
  - a grace penalty may be charged after end + graceConfig.period
  - principal repayment is forwarded to Moolah and booked into principalRepaid
  - when principal is fully repaid, the position is removed from the user's array (swap-and-pop removal).

Appendix: Relayer Routing

Interest and penalties are treated as revenue intended to be supplied into the Moolah vault, but supplies are batched to avoid sub-minimum deposits.

For interest and penalty paid in the loan token, the broker increases allowance to RELAYER and calls RELAYER.supplyToVault(amount).

For interest paid in LISTA:

- the user transfers LISTA to the relayer,
- the broker computes the discounted loan-token-equivalent interest amount,
- and pulls loan tokens from the relayer into the broker via RELAYER.transferLoan(interestAmount) so the repayment can be executed in the loan token domain

The batching/threshold logic for when revenue is actually supplied into Moolah is implemented in the relayer.

Privileged Functions

- setMarketId
- updateFixedTermAndRate
- setMaxFixedLoanPositions
- setBorrowPaused
- setGraceConfig
- setListaDiscountRate
- pause
- unpause

Core Invariants:

INV 1: borrow is not allowed if user has debt

INV 2: Each borrow action must create a new fixed term loan

INV 3: Interest is always denominated in LOAN_TOKEN

INV 4: During repayment, it must always favor interest repayment -> penalty -> principal

INV 5: The aggregate of remainingPrincipal for all positions from a user must exactly match the borrowed amount on moolah for this user

INV 6: Setting lastRepaidTime for ACCRUE fixed term loans requires interestRepaid to be reset

INV 7: A position must be removed if principalRepaid >= principal

INV 8: setMarketId is only called once

INV 9: repayInterestWithLista must only allow for repaying interest and no penalty/principal

INV 10: A user can have maximum of maxFixedLoanPositions number of fixed positions open at a time

INV 11: Borrowing should not be possible when borrowPaused is true

Issue_04	Interest repayment with LISTA is fundamentally broken
Severity	**High**
Description	Whenever a repayment with LISTA is executed it will receive the LOAN_TOKEN from the relayer and then continue with the internal _repay function. This function however still transfers the full LOAN_TOKEN amount from the user, effectively double charging + leaving LOAN_TOKEN from the relayer permanently locked within the CreditBroker.
Recommendations	Consider developing a second internal _repay function (_repayWithLista) which solely handles the repayment via the received LOAN_TOKEN amount from the relayer. It should not touch any penalty / interest to reduce complexity.
Comments / Resolution	Resolved. Now only (repayInterestAmt - receivedInterest) is pulled from the user.

Issue_05	_validatePosition breaks in different edge-cases
Severity	**High**
Description	The _validatePosition function freezes fully if a user has 2 >= positions below minLoan because repaying one will still result in the revert of the other. There are two occasions when this can happen: - Increase of minLoanValue on Moolah - Oracle price decrease (depeg of the stablecoin) The first even enables a directional attack-path: - frontrun minLoanValue increase - repay two positions of a user so they are both below new minLoanValue - DoS
Recommendations	Consider removing this check and instead implementing a check which is isolated to the CreditBroker state.
Comments / Resolution	Acknowledged

Issue_06	Penalty can be gamed via two-step repayments
Severity	**High**
Description	The penalty calculation within the CreditBrokerMath contract includes the accrued interest before the deduction during the _repay function: > *uint256 debt = remainingPrincipal + accruedInterest;* Users that want to repay their whole interest + penalty + principal can do this in two steps instead of one to drastically reduce the paid penalty because penalty is only applied if actual principal is repaid. Therefore, users can repay only interest during the first transaction which then results in decreased penalty during the second transaction because the interest has been zero'd out.
Recommendations	Consider either deducting the paid interest from the previous accounting during _repay (which also fixes another issue) or refactoring the logic fully.
Comments / Resolution	Resolved, it is now expected that a penalized position is only repaid in full, which prevents gaming in such a scenario. It has to be noted that a user can still repay interest just one block before a position becomes penalized.

Issue_07	Liquidations are not supported
Severity	**High**
Description	Currently, liquidations are not supported due to the revert within the CreditBroker. Furthermore, if they would ever be supported in the future for this system, they are broken: - liquidations could bring positions below minLoanValue which may break CreditBroker assumptions - liquidations will desync state between CreditBroker and Moolah - Liquidation control-flow does not override price (for broker callpath) in seizedAmount calculation, leaving repaid shares understated. Can even result in price = 0 - Credit tokens may be transferred to ineligible addresses (liquidator) It is furthermore generally an issue that liquidations are not supported because the interest may result in a position becoming above LTV of 100 and/or the LOAN_TOKEN may be borrowed during times of depeg and suddenly increases in value which also means an excess of LTV.
Recommendations	Consider implementing liquidations properly and ensuring a full audit of the liquidation mechanism.
Comments / Resolution	Acknowledged, this will be subject to a different engagement.

Issue_08	Users can steal value from the Relayer by exploiting the Lista discount through sandwiched repayments
Severity	**Medium**
Description	In case the interest repayment is made using Lista token, the users obtain a discount. This discount is implemented by the BrokerRelayer paying the rest of the interest on behalf of the user. So if 100 was the total interest and 20% was the discount rate, then the user only has to pay 80 worth of Lista tokens and the relayer will repay 100 interest on behalf of the user in exchange for the 80 worth Lista tokens. Since the interest repayments eventually go to the vault shareholders, a user can exploit this and drain the Relayer by making a huge deposit to the vault and then making the interest repayment using Lista. Depending on the % of shares the user obtains through the large deposit, the user can earn up to discountRate profit instantly Eg: discountRate = 20 vaultShares = 1000 vaultAssets = 1000 interestRepayment = 100 attacker makes a deposit of 10000 to vault and repays the interest using lista, attacker pays 80 worth lista and obtains (100 * 10000 / 11000) == 90 worth of interest. hence a gain of 10
Recommendations	Consider disallowing same block withdrawals in the vault, this should disable large flash loan deposits before repayments
Comments / Resolution	Acknowledged.

Issue_09	Race condition due to updateFixedTermAndRate
Severity	**Medium**
Description	The updateFixedTermAndRate function allows governance to change parameters such as APR and duration. If changed just before a user's borrow execution, this might result in a large disadvantage for the user ie. higher APR than expected.
Recommendations	Consider communicating such a change with the community upfront.
Comments / Resolution	Resolved. It has to be noted that now existing terms cannot be removed which means if the manager makes a setting mistake, users can always borrow using that term.

Issue_10	getUserTotalDebt does not include penalty
Severity	**Medium**
Description	Currently, the getUserTotalDebt function includes principal and interest in the debt but fails to include penalties. This means the health check in Moolah is inaccurate and lets users borrow more than they should be able to.
Recommendations	Consider including any penalty as well.
Comments / Resolution	Acknowledged.

Issue_11	Race condition due to decrease of listaDiscountRate can result in users repaying less interest than expected
Severity	**Medium**
Description	Whenever governance decreases the listaDiscountRate variable it means that a LISTA repayment will be equivalent to less interest than before. This means users will essentially achieve a less favorable outcome.
Recommendations	Consider making this function a two-step process with pending and setting.
Comments / Resolution	Acknowledged.

Issue_12	Change of graceConfig.period may penalize users unexpected because of a lack of caching during creation
Severity	**Medium**
Description	The graceConfig.period variable is used to determine whether a position is being penalized. If this value is ever decreased, it can happen that positions are penalized unexpectedly, because it is not tied to a position.
Recommendations	Consider caching this value whenever a new position is created into the FixedLoanPosition struct and then using this value to determine whether a position is penalized.
Comments / Resolution	Acknowledged.

Issue_13	Multiple issues occur if interest is enabled on Moolah
Severity	**Medium**
Description	Whenever the CreditBroker is used with a market that has interest, multiple edge-cases can occur, including but not limited to: - Desynchronization due to accrual during repayment may result in overestimating required assets and DoS repayment - Incorrect debt determination for various operations - Desynchronization between principal and real debt - DoS of repayment due to truncation of amount -> share conversation, leaving dust which reverts on minLoan check
Recommendations	Consider not enabling interest in Moolah and ensuring that a market which is tied to a CreditBroker has always a share <-> asset ratio of 1.
Comments / Resolution	Acknowledged. Interest will never be enabled.

Issue_14	isPositionPenalized doesn't handle period == 0 case
Severity	**Low**
Description	In case block.timestamp is greater than position.end + graceConfig.period, isPositionPenalized always returns true. But in case graceConfig.period was 0, the position won't actually be penalized during repayments causing a mismatch between the view function and the actual execution
Recommendations	Return false in case graceConfig.period is 0
Comments / Resolution	Resolved, logic has been changed which assumes that if period = 0, positions become instantly penalized.

Issue_15	UPFRONT interest is not included in health check during initial phase
Severity	**Low**
Description	Within UPFRONT positions, the full interest is payable as soon as the noInterestPeriod has surpassed. This interest is not reflected at all during the noInterestPeriod, resulting in immediate and potentially large bad-debt positions whenever the cliff is reached.
Recommendations	Consider whether it makes sense to include this interest into the health check even if its not yet applicable (and potentially may never be if repaid fully before)
Comments / Resolution	Acknowledged.

Issue_16	Lack of input validation within supply and withdraw control-flows
Severity	**Low**
Description	During supply and withdrawal of collateral, the external call to Moolah uses the marketParams parameter which is user provided. The only input validation here is the cross-check against the COLLATERAL_TOKEN: *require(marketParams.collateralToken == COLLATERAL_TOKEN, "broker/invalid-collateral-token");* Side-effects can range from supplying to other markets/withdrawing from other markets with the COLLATERAL_TOKEN to more advanced attack scenarios. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider increasing the strictness for input validation.
Comments / Resolution	Resolved.

Issue_17	setGraceConfig does not allow only changing noInterestPeriod only
Severity	**Low**
Description	The setGraceConfig function has the following check: *require(graceConfig.period != period || graceConfig.penaltyRate != penaltyRate, "broker/same-value-provided");* This means it will be impossible to only change noInterestPeriod while keeping the other parameters equal.
Recommendations	Consider removing this check.
Comments / Resolution	Resolved.

Issue_18	_tryWithdrawAndBurnDebt may result in lower collateralization than expected
Severity	Low
Description	The _tryWithdrawAndBurnDebt function is triggered before any borrow execution and essentially forces users to withdraw their collateral to offset the current debt. First of all, this mechanism does not work in the following scenarios: - If the debt is higher than the collateral value - If the debt withdrawal would result in the position to become undercollateralized However, the biggest problem with this implementation is that it can result in a lower collateralization as expected by the user, potentially even in a liquidatable state.
Recommendations	This risk is inherent from the new implementation.
Comments / Resolution	

Issue_19	repayInterestWithLista can be used to repay normal principal without interest
Severity	Low
Description	The repayInterestWithLista function requires listaAmount > 0 but in case there is no outstanding interest it will clamp listaAmount to zero and follows the normal control-flow of principal and penalty repayment.
Recommendations	Consider reverting if there is no outstanding interest.
Comments / Resolution	

Issue_20	Inconsistency between isPositionPenalized and _isPositionPenalized
Severity	**Low**
Description	The isPositionPenalized function returns false if period = 0 and if rate = 0. However, during the _repay function, it consults the _isPositionPenalized function which marks a position as penalized even if period = 0 and rate = 0, it will simply ask for full repayment (even if penalty = 0). This creates a discrepancy between both definitions and allows for borrowing (ie. isUserPenalized early return) while a position may still require a full repayment as per _isPositionPenalized definition.
Recommendations	Considering marking a position as penalized within the view isPositionPenalized function even if the rate is zero. The same counts for isUserPenalized.
Comments / Resolution	

Issue_21	Liquidation reverts unexpectedly because of incorrect function signature
Severity	**Informational**
Description	Liquidations are not supported in the current system and this is enforced by the CreditBroker contract reverting inside the liquidate function. But the function signature is incorrect and hence the liquidation call from Moolah will revert due to lack of function selector match rather than due to the revert inside the liquidate function
Recommendations	Change the function signature of liquidate in CreditBroker to function liquidate(Id id, address borrower)
Comments / Resolution	Resolved

Issue_22	OOG concerns within updateFixedTermAndRate
Severity	**Informational**
Description	The fixedTermAndRate function may run OOG if the to be determined termID is near the end of the array and there are many terms.
Recommendations	Consider limiting the amount of possible terms.
Comments / Resolution	Acknowledged.

Issue_23	Unused variables
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers: *modifier onlyMoolah()*
Recommendations	Consider removing these unused variables.
Comments / Resolution	Resolved.

CreditBrokerMath

CreditBrokerMath is a stateless calculation library that is linked into a CreditBroker implementation to produce the accounting numbers the broker relies on:

- user debt totals
- fixed-loan interest accrual
- penalties
- LISTA↔loan-token conversions
- other functionality (unused)

The library does not custody tokens, write storage, or enforce permissions. Instead, it executes inside the caller's context and assumes address(this) is an ICreditBroker that exposes LOAN_TOKEN(), COLLATERAL_TOKEN(), MARKET_ID(), MOOLAH(), and userFixedPositions(user). Any state transitions (repay, liquidation settlement, vault supply) are performed by the broker contract; this library only computes values and (in one helper) returns an updated in-memory copy of a fixed position for the caller to persist.

Multiple functionalities remain unused in the context of the CreditBroker and should be fully reviewed in the context of other contracts where these are intended to be invoked.

Appendix: Fixed-loan interest models

Accrued interest (ACCRUE_INTEREST):

Interest accrues linearly over time from the last repayment anchor:

- elapsed = min(now, end) – min(lastRepaidTime, end)
- interest = ceil(remainingPrincipal × aprPerSecond(apr) × elapsed / RATE_SCALE)
- aprPerSecond(apr) is ceil((apr – RATE_SCALE) / 365 days) and is zero when apr <= RATE_SCALE.

Upfront interest (UPFRONT_INTEREST):

Interest is computed for the entire term once the no-interest window has passed:

- if now <= noInterestUntil, interest is 0

- totalInterest = ceil(principal × ceil((apr – RATE_SCALE) × term / 365 days) / RATE_SCALE)

Appendix: Penalty Application

A penalty to a position is applied according to the following rules:

- No penalty when graceConfig.period == 0 or now <= endTime + graceConfig.period.
- Compute debt = remainingPrincipal + accruedInterest.
- Compute maxPenalty = ceil(debt × penaltyRate / RATE_SCALE).
- If the proposed repayment amount is large enough to cover remainingPrincipal + maxPenalty, charge maxPenalty; otherwise charge ceil(repayAmt × penaltyRate / RATE_SCALE).

Appendix: Lista Conversion

The Broker allows interest repayment via LISTA at a discounted effective rate, while keeping conversions deterministic under 8-decimal pricing.

- Discounted interest = ceil(remainingInterest × (RATE_SCALE – discountRate) / RATE_SCALE).
- Max LISTA accepted = ceil(discountedInterest × 1e8 / listaPrice).
- Interest implied by LISTA payment:
  - loanTokenAmount = floor(listaAmount × listaPrice / 1e8)
  - interestAmount = floor(loanTokenAmount × RATE_SCALE / (RATE_SCALE – discountRate))

Privileged Functions

- none

Core Invariants:

INV 1: UPFRONT_INTEREST positions must display full interest only after position.noInterestUntil

INV 2: A penalty is only ever charged if a position is not repaid in time

INV 3: Interest is only accrued till the term end

INV 4: Repaid amount via Lista doesn't exceed pending interest

Issue_24	Lack of normalization during repayment of interest with LISTA
Severity	**High**
Description	Currently, the math within the repayment with LISTA control-flow is as follows: ceil([accruedInterest * ceil([1e27 - discountRate]) / 1e27] * 1e8 / listaPriceIn1e8] floor(floor([listaAmount * listaPrice / 1e8]) * 1e27 / (1e27 - discountRate]) The following things are missing: - Oracle assumes that LOAN_TOKEN will always be in USD, this can be incorrect - LOAN_TOKEN may have != 18 decimals If one of these things is applicable in the future (depending on usage for different tokens), the math will be incorrect and it will break the logic.
Recommendations	Consider implementing decimal normalization and ensuring that the oracle returns the LISTA price denominated in the LOAN_TOKEN
Comments / Resolution	Resolved.

Issue_25	maxPenalty clamping is incorrect
Severity	**High**
Description	The maxPenalty clamping logic is currently inaccurate as it does not actually prevent a penalty above maxPenalty. We will demonstrate this with a simple example: - remainingPrincipal = 100 - interest = 0 - penaltyRate = 15% - maxPenalty = 100 * 15% - maxPenalty = 15 - maxRepayable = 115 - we repay only 114 (**below maxRepayable**) - 114 * 15% = 18 We are above maxPenalty
Recommendations	Consider changing the condition check or clamping the penalty appropriately.
Comments / Resolution	Resolved.

Issue_26	Incorrect rounding direction for max LISTA calculation can result in repaying principal with LISTA - instead of interest only
Severity	**Medium**
Description	The getMaxListaForInterestRepay function for interest rounds up but it should round down to ensure one cannot provide more LISTA to repay part of principal: - accruedInterest = 9 (in smallest loan-token units) - discountRate = 0.2e27 → (S-d)/S = 0.8 - interestAfterDiscount = ceil(9 * 0.8) = ceil(7.2) = 8 - maxLista = ceil(8 * 1e8 / 1e8) = 8 - interestAmountFromLista = floor(8 / 0.8) = floor(10) = 10 Result: interestAmountFromLista(maxLista) = 10 > 9
Recommendations	Consider clamping the calculated interest from lista amount to maxInterest
Comments / Resolution	Resolved.

Issue_27	Unused functions
Severity	**Informational**
Description	Functions which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. This library exposes various different helper functions which are unused.
Recommendations	Consider removing all functions that are not actively used by the CreditBroker.
Comments / Resolution	Resolved.

Issue_28	Calculation of penalty based on initial interest
Severity	**Informational**
Description	The penalty is created based on the interest before the function execution. This is systematically correct if it is indeed intended to charge a penalty based on this interest. However, the variable naming: > accruedInterest is not fully accurate as it would need to be accruedInterestBeforeRepayment. Furthermore, it opens up ambiguous gaming the penalty calculation as mentioned in another issue.
Recommendations	Consider calculating the penalty solely on the remaining principal.
Comments / Resolution	Acknowledged.

Issue_29	Typographical error
Severity	**Informational**
Description	The following typographical error is present: *@dev Calculate the max LISTA amount accpetable for repaying interest*
Recommendations	Consider fixing the spelling mistake.
Comments / Resolution	Resolved.

Issue_30	getPositionDebt function is unused
Severity	**Informational**
Description	During the resolution round, the getPositionDebt function was implemented. This function is however unused in any context and is also not directly exposed since the CreditBrokerMath is a library and no contract.
Recommendations	Consider removing that function or implementing a view-only path via the broker.
Comments / Resolution	

Issue_31	previewRepayFixedLoanPosition incorrectly assumes that full repayment is needed only if penaltyRate is > 0
Severity	**Low**
Description	If the calculated penalty is 0, previewRepayFixedLoanPosition assumes that a full repayment is not needed. This is incorrect because even when penaltyRate is 0 (and hence 0 penalty as well), full repayment is necessary if graceConfig.period is non-zero and the current timestamp is greater than end + period
Recommendations	Skip full repayment only if the position is not penalized
Comments / Resolution	

Lending

Moolah

The Moolah contract is a vault-style lending protocol which incorporates different markets in the same contract and allows users to supply tokens which can then be borrowed by other users with sufficient backed collateral.

The main focus of the review on this contract are the control-flows outgoing from the CreditBroker contract for the loan-specific market which incorporates the fixed term lending product with credit tokens.

The control-flows that have been reviewed in this context were mainly limited to supplyCollateral/withdrawCollateral/borrow/repay. All other functionalities remain unused in the context of the fixed term lending product with credit tokens.

It is expected that the CreditBroker is set as provider for the specific market via the setProvider function and likewise as broker for the specific market via the setMarketBrokers function - this ensures proper access control such that only the CreditBroker contract can invoke supplyCollateral/withdrawCollateral/repay/borrow for the specified market. Furthermore it is expected that there is no interest accrual for this specific market as the interest is outsourced to the CreditBroker in the form of fixed interest.

Privileged Functions

- enableIRM
- enableLltv
- setFee
- setDefaultMarketFee
- setFeeRecipient
- setMinLoanValue
- batchToggleLiquidationWhitelist
- setWhitelist
- setVaultBlacklist
- setProvider
- setFlashloanTokenBlacklist
- setMarketBroker
- createMarket
- pause
- unpause

Core Invariants (related to Broker):

INV 1: Only the CreditBroker can call borrow with itself being the recipient

INV 2: Only the CreditBroker can call repay

INV 3: Only the CreditBroker can call supplyCollateral

INV 4: Only the CreditBroker can call withdrawCollateral with itself being the recipient

INV 5: Liquidations are currently not supported

INV 6: The marketID in context must not have any interest accrual

INV 7: share:asset ratio for underlying markets of CreditBrokers should remain 1

Issue_32	Withdrawal or borrowing of loan tokens can be griefed
Severity	**Medium**
Description	**In a previous audit, Bailsec has raised the following issue (Known Morpho Issues):** "With the current implementation of Morpho interest does not accrue within the same block and one is not disallowed to borrow and repay in the same block. These two properties open a griefing attack that: 1. Borrowing liquidity can be blocked. 2. Withdrawing liquidity by the suppliers can be blocked. To perform such attacks the attacker would only need to: 1. Pay for transaction gas fees. 2. Have enough collateral to be able to borrow all the remaining liquidity. The attack works as follows assume T1 is the transaction the honest actor wants to submit to either borrow or withdraw liquidity, then the attacker frontruns T1 with T0 which is the transaction to remove/borrow all the liquidity by providing enough collateral and back runs with T2 which is the transaction that would repay the loan and withdraws the collateral. Overall, the attacker would get all its collateral back but indeed would need to spend on gas and depending on the market state have enough liquidity to begin with. Griefing the borrow end point can be tolerable as the honest actor would still have access to their tokens. But blocking a supplier to withdraw their loan tokens is one that is more important."
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_33	Users can take advantage of low liquidity markets to inflate the interest rate
Severity	**Medium**
Description	**In a previous audit, Bailsec has raised the following issue (Known Morpho Issues):** "Morpho Blue is meant to work with stateful Interest Rate Models (IRM) whenever _accrueInterest() is called, it calls borrowRate() of the IRM contract. This will adjust the market's interest rate based on the current state of the market. For example, AdaptiveCurveIrm.sol adjusts the interest rate based on the market's current utilization rate. However, this stateful implementation will always call borrowRate() and adjust the interest rate, even when it should not. For instance, in AdaptiveCurveIrm.sol, an attacker can manipulate the market's utilization rate as such: • Create market with a legitimate loanToken, collateralToken, oracle and the IRM as AdaptiveCurveIrm.sol. • Call supply() to supply 1 wei of loanToken to the market. • Call supplyCollateral() to give himself some collateral. • Call borrow() to borrow the 1 wei of loanToken. • Now, the market's utilization rate is 100%. • Afterwards, if no one supplies any loanToken to the market for a long period of time, Adaptive CurveIrm.sol will aggressively increase the market's interest rate. This is problematic as Morpho Blue's interest compounds based on e^x. As such, when borrowRate (the interest rate) increases, interest will grow at an exponential rate, which could cause the market's totalSupplyAssets and totalBorrowAssets to become extremely huge. This creates a few issues: 1. The market will have a huge amount of unclearable bad debt: Should a large amount of interest accrue, totalBorrowAssets will be

	extremely large, even though totalBorrowShares is only 1e6 shares. Half of totalBorrowAssets would have actually accrued to the other 1e6 virtual shares. As such, after liquidating the attacker's 1e6 shares, half of totalBorrowAssets will still remain in the market as un-clearable bad debt. 2. The market will permanently have a high interest rate: As mentioned above, AdaptiveCurveIrm.sol aggressively increased the market's interest rate while there was only 1 wei supplied and borrowed in the market, causing utilization to be 100%. If other lenders decide to supply loanToken to the market, borrowers would still be discouraged from borrowing for an extended period of time as AdaptiveCurveIrm.sol would have to adjust the market's interest rate back down. 3. Users who call supply() with a small amount of assets might lose funds: If totalSupplyAssets is sufficiently large compared to totalSupplyShares, the market's shares to assets ratio will be huge. This will cause the following the share calculation in supply() to round down to 0: *if (assets > 0) shares = assets.toSharesDown(market[id].totalSupplyAssets, market[id].totalSupplyShares);* *else assets = shares.toAssetsUp(market[id].totalSupplyAssets, market[id].totalSupplyShares);* Should this occur, the user will receive 0 shares when depositing assets, resulting in a loss of funds. Partially fixed. The attack can still be executed with minLoanValue instead of 1 wei. If the attacker is the only user in a market, this exploit is unavoidable.”
Recommendations	**Since this risk has been acknowledged before, this issue has**

	only been added to provide a full comprehensive report. No actions needed.
Comments / Resolution	Acknowledged.

Issue_34	Bad debt socialization can be skipped
Severity	**Medium**
Description	**In a previous audit, Bailsec has raised the following issue:** "In the liquidate function the liquidator can specify the exact amount of collateral to seize from the position (seizedAssets) and the corresponding borrowed asset/shares is calculated. In the case of a bad debt, where the users collateral is not enough to fully cover their loan/borrowed assets, the remaining assets are removed from the total supplied assets, socializing the bad debt among all current lenders. The issue with this is that the bad debt is only socialized when the liquidator completely seizes all the borrowers collateral assets (i.e. *position[id][borrower].collateral == 0*) In a case where a liquidator is incentivized to prevent bad debt socialization (a case where liquidator also has supply shares in the market) or a liquidator simply acting in bad faith. Such liquidator can prevent the debt socialization by calling the liquidate function with a seizedAssets amount 1 wei less than the borrower's collateral, bypassing the bad debt check and possibly leading to insolvency issues." Please note that this issue was only partially resolved because If a borrower gets liquidated with an amount of remaining bad debt higher than the minimum loan amount, the liquidator can still pull this attack because _isHealthyAfterLiquidate will return true. Therefore, the current contract includes this issue as a medium severity.
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**

Comments / Resolution	Acknowledged.

Issue_35	Unpausing the protocol will result in instant liquidations
Severity	**Medium**
Description	**In a previous audit, Bailsec has raised the following issue:** "The Moolah contract is pausable by the PAUSER role. In case the protocol is paused for a long time, borrower's collateral can fall in price, or the position can become unhealthy due to interest accrual, which will result in positions becoming liquidatable without a way for borrowers to add collateral or repay their loans. When the protocol is unpaused by the MANAGER, instant liquidations will occur resulting in losses for the borrowers."
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_36	minLoanValue is bypassable via withdrawals
Severity	Medium
Description	In a previous audit, Bailsec has raised the following issue: “The introduced _checkSupplyAssets function prevents users from having below minLoanValue worth of assets. However this check is not done after withdraw , which means users can withdraw any amount after supplying , possibly leaving less than minLoanValue worth of assets in their position.”
Recommendations	Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.
Comments / Resolution	Acknowledged.

Issue_37	Flash Loan Attack via ERC-4626 Share Token Manipulation
Severity	Medium
Description	**In a previous audit, Bailsec has raised the following issue:** "A known vulnerability in lending protocols allows attackers to manipulate the value of ERC-4626 share tokens using flash loans. In the context of the Moolah protocol, this can lead to bad debt if an ERC-4626 share token is used as collateral and most of its supply is deposited in Moolah. If exploited, this attack can artificially deflate the price of a collateral token, allowing an attacker to borrow more than they should and leave the protocol with unbacked debt. **Requirements:** - A Moolah market that accepts an ERC-4626 share token as collateral - Most (or all) of the share token supply is deposited in the Moolah market **Attack steps**: 1. The attacker acquires some of the ERC-4626 share token and uses it as collateral to borrow another asset from Moolah. 2. The attacker takes a flash loan of the ERC-4626 share token from the market, borrowing the entire token supply. 3. They redeem all the share tokens through the ERC-4626 vault, draining its underlying assets and resetting the share price. 4. They then mint new share tokens using fewer underlying assets (now that the share price is 1). 5. The attacker repays the flash loan with the newly minted shares. 6. The share price has now been reset, making the original collateral worth less than the borrowed amount, resulting in protocol bad debt. Moolah allows permissionless market creation, meaning anyone can create a market that accepts ERC-4626 tokens as collateral. If most of a token's supply ends up the protocol, the conditions for the exploit are met."
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions**

	needed.
Comments / Resolution	Acknowledged.

Issue_38	Deviation in oracle price could lead to arbitrage in high lltv markets
Severity	**Medium**
Description	**In a previous audit, Bailsec has raised the following issue (Known Morpho Issues):** "In Morpho Blue, the maximum amount a user can borrow is calculated with the conversion rate between loanToken and collateralToken returned by an oracle. However, all price oracles are susceptible to front-running as their prices tend to lag behind an asset's real-time price. More specifically: Chainlink oracles are updated after the change in price crosses a deviation threshold, (eg. 2.5% in ETH / USD), which means a price feed could return a value slightly smaller/larger than an asset's actual price under normal conditions. Uniwap V3 TWAP returns the average price over the past X number of blocks, which means it will always lag behind the real-time price. An attacker could exploit the difference between the price reported by an oracle and the asset's actual price to gain a profit by front-running the oracle's price update. For Morpho Blue, this becomes profitable when the price deviation is sufficiently large for an attacker to open positions that become bad debt.

	The likelihood of this condition becoming true is significantly increased when ChainlinkOracle.sol is used as the market's oracle with multiple Chainlink price feeds.”
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_39	Suppliers can be tricked into supplying more
Severity	Medium
Description	**In a previous audit, Bailsec has raised the following issue (Known Morpho Issues):** "The supply and withdraw functions can increase the supply share price (totalSupplyAssets / totalSupplyShares). If a depositor uses the shares parameter in supply to specify how many assets they want to supply they can be tricked into supplying more assets than they want. It's easy to inflate the supply share price by 100x through a combination of a single supply of 100 assets and then withdrawing all shares without receiving any assets in return. The reason is that in withdraw we compute the assets to be received as *assets = shares.toAssetsUp(market[id].totalSupplyAssets, market[id].totalSupplyShares);.* Note that assets can be zero and the withdraw essentially becomes a pure burn function. Example: • A new market is created. • The victim tries to supply 1 assets at the initial share price of 1e-6 and specifies supply(shares=1e6). They have already given max approval to the contract because they already supplied the same asset to another market. • The attacker wants to borrow a lot of the loan token and therefore

	targets the victim. They frontrun the victim by supply(assets=100) and a sequence of withdraw() functions such that totalSupplyShares = 0 and totalSupplyAssets = 100. The new supply share price increased 100x. • The victim's transaction is minted and they use the new supply share price and mint 100x more tokens than intended (possible because of the max approval). • The attacker borrows all the assets. • The victim is temporarily locked out of that asset. They cannot withdraw again because of the liquidity crunch (it is borrowed by the attacker). Partially fixed. Now there's a minimum amount check on supply, but that check should also be applied on withdraw. The attack can still be executed by supplying the minimum amount allowed, and repeatedly call withdraw to bring totalSupplyShares to zero."
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_40	Any oracle update with sufficiently big price decline can be sandwiched to extract value from the protocol
Severity	Medium
Description	**In a previous audit, Bailsec has raised the following issue (Known Morpho Issues):** "Whenever oracle update provides substantial enough price decline the fixed nature of liquidation incentive along with the fixed LLTV for the market provides for the ability to artificially create bad debt and immediately liquidate it, stealing from protocol depositors. For highly volatile collateral it's possible to sandwich Oracle update transaction (tx2), creating min collateralized loan before (tx1) and liquidating it right after, withdrawing all the collateral (tx3). As liquidation pays the fixed incentive and socialize the resulting bad debt, if any, all the cases when bad debt can be created on exactly one Oracle update (i.e everywhere when it's sufficiently volatile asset respect to LLTV), this can be gamed, as the attacker will pocket the difference between debt and collateral valuation, which they receive in full via liquidate-withdraw collateral sequence in tx3. Notice that initial LLTV setting might be fine for the collateral volatility at that moment, but as particular collateral/asset pair might become substantially more volatile (due to any developments in collateral, asset or changes of the overall market state), while there is no way to prohibit using LLTV once enabled. Proof of Concept: • Morpho instance for stablecoins was launched with USDC collateral and USDT asset allowed, and with LLTV set included competitive reading of 95%, over time it gained TVL, USDC and USDT is now traded 1:1. Liquidation incentive factor for the market is liquidationIncentiveFactor = 1.0 / (1 - 0.3 * (1 - 0.95)) = 1.01522. • There was a shift in USDC reserves valuation approach, and updated reserves figures are priced in

	sharply via new Oracle reading of 0.9136 USDT per USDC. • Bob the attacker front runs the Oracle update transaction (tx2) with the borrowing of USDT 0.95m having provided USDC 1m (tx1, and for simplicity we ignore dust adjustments in the numbers where they might be needed as they don't affect the overall picture). • Bob back-runs tx2 with liquidation of the own loan (tx3), repaying USDT 0.9m of the USDT 0.95m debt. Since the price was updated, with repaidAssets = USDT 0.9m he will have seizedAssets = USDC 0.9m * 1.01522 / 0.9136 = USDC 1m. • Bob regained all the collateral and pocketed USDT 0.05m, which was written off the deposits as bad debt. So, an Attacker can steal principal funds from the protocol by artificially creating bad debt. This is a permanent loss for market lenders, its severity can be estimated as high. The probability of this can be estimated as medium as collateral volatility is not fixed in any way and sharp downside movements will happen from time to time in a substantial share of all the markets. The prerequisite is that initially chosen LLTV does not fully guarantee the absence of bad debt after one Oracle update. This effectively means that LLTV has to be updated to a lower value, but it is fixed within the market and such changes are usually subtle enough (as compared to more substantially scrutinized initial settings), so, once that happens, there is substantial probability that there will be a window of opportunity for attackers, the period when LLTV of a big enough market being outdated and too high, but there was no communication about that and the marker being actively used."
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_41	Lack of sanity checks for createMarket
Severity	Low
Description	**In a previous audit, Bailsec has raised the following issue (Known Morpho Issues):** "The current implementation of createMarket is not covering all the marketParams attributes with sanity checks. This could allow the creation of broken markets or markets that do not make sense. Moolah should consider implementing the following sanity checks: • marketParams.loanToken != address(0) && loanToken != collateralToken. • marketParams.collateralToken != address(0) (the collateralToken != loanToken check is implicit because of the previous check). • marketParams.oracle.price() returns a valid answer and does not revert. • marketParams.irm is working as expected without reverting. Partially fixed. Oracle.peek() could return 0 as price instead of reverting, but that is not a healthy oracle. Recommendation is to validate the returned price is > 0"
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_42	Function getPrice only works when oracle prices have the same decimals
Severity	Low
Description	**In a previous audit, Bailsec has raised the following issue:** "The new getPrice function works as following: 1. It queries the decimals of the collateral and loan tokens 2. It queries the oracle price of both tokens 3. It calculates the scaled relative price between the two tokens by adjusting the oracle prices with a scaling factor. This process only works correctly when the oracle prices have the exact same decimals. While this is usually the case with Chainlink and Binance feeds, there are some price feeds that implement different decimals than the rest. Given that market creation is permissionless on Moolah, it's expected that some tokens are going to be used which have different decimals on their feeds. A market that uses these tokens will cause the getPrice function to be broken and cause a loss of funds to its users."
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_43	Virtual supply shares steal interest
Severity	Low
Description	In a previous audit, Bailsec has raised the following issue (Known Morpho Issues): “The virtual supply shares, that are not owned by anyone, earn interest in _accrueInterest. This interest is stolen from the actual suppliers which leads to loss of interest funds for users. Note that while the initial share price of 1e6 might make it seem like the virtual shares can be ignored, one can increase the supply share price and the virtual shares will have a bigger claim on the total asset percentage.”
Recommendations	Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.
Comments / Resolution	Acknowledged.

Issue_44	Virtual borrow shares accrue interest and lead to bad debt
Severity	Low
Description	In a previous audit, Bailsec has raised the following issue (Known Morpho Issues): “The virtual borrow shares, that are not owned by anyone, earn interest in _accrueInterest. This interest keeps compounding and cannot be repaid as the virtual borrow shares are not owned by anyone. As the withdrawable funds are computed as supplyAssets - borrowAssets, the borrow shares' assets equivalent leads to a reduction in withdrawable funds, basically bad debt. Note that while the initial borrow shares only account for 1 asset, this can be increased by increasing the share price. The share price can be inflated arbitrarily high.”
Recommendations	Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.
Comments / Resolution	Acknowledged.

Issue_45	Users that borrow as much as possible (up to the LLTV threshold) will be liquidable in the very next block
Severity	Low
Description	**In a previous audit, Bailsec has raised the following issue (Known Morpho Issues):** "Unlike protocols like Aave where they have both LTV (loan to value) and LT (liquidation threshold), the current implementation of Morpho offers just the LLTV parameter (Liquidation Loan-To-Value) without any buffer between the amount of debt that the user can take (given a collateral) and the liquidation threshold. This means that if a user borrows as much as it can borrow (given a collateral and LLTV), such user will be fully liquidable in the very next block as soon as the interest accrual can be triggered."
Recommendations	**Since this risk has been acknowledged before, this issue has only been added to provide a full comprehensive report. No actions needed.**
Comments / Resolution	Acknowledged.

Issue_46	Flashloan of creditToken may create unexpected state
Severity	**Informational**
Description	Currently it is possible to flashloan any token if it is not blacklisted. This is also true for the creditToken which is not meant to be transferred to users that are not eligible as such. With a flashloan, an ineligible user can become the owner of a creditToken for the duration of the transaction which is in itself an invalid state. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider blacklisting the creditToken for flashloans
Comments / Resolution	Acknowledged

Issue_47	Risk of Incorrect carket creation due to 100% LLTV Configuration being enabled
Severity	**Informational**
Description	Creating credit token markets requires temporarily enabling 100% LLTV in Moolah. Since market creation can be permissionless, this configuration may be abused if left enabled beyond the intended deployment phase. If permissionless market creation is active while 100% LLTV is enabled, users could create markets that are intended to require a lower LLTV, potentially breaking core protocol assumptions and leading to unsafe market configurations.
Recommendations	Ensure that 100% LLTV is disabled once all intended credit token markets have been deployed.
Comments / Resolution	

Relayer

CreditBrokerInterestRelayer

The CreditBrokerInterestRelayer contract is a dedicated relaying module which intermediates between whitelisted CreditBroker instances and a designated Moolah vault, ensuring that interest and penalty revenues collected by brokers are aggregated and only supplied into the vault once they exceed the market's minimum position size constraint. It maintains an internal accounting (supplyAmount) of loan tokens earmarked for vault supply, supports broker-triggered vault deposits in thresholded batches, and also exposes a controlled "loan token release" mechanism that allows brokers to pull loan tokens from the relayer when performing discounted interest repayment flows (where users pay LISTA off-path and the relayer settles the broker in loan tokens).

The relayer is initialized with the Moolah address, a single vault receiver address, and a single ERC20 token that represents the loan token for all whitelisted brokers. Brokers are explicitly whitelisted by the relayer's MANAGER role and are additionally validated at whitelist time by requiring that the broker's reported LOAN_TOKEN() equals the relayer's configured token, preventing brokers of other loan assets from interacting with this relayer instance.

Operationally, brokers call supplyToVault(amount) to forward interest/penalty proceeds into the relayer. The relayer transfers amount from the broker into itself and increments supplyAmount. It then derives the applicable minLoan threshold from Moolah using the calling broker's market id (ICreditBrokerBase(msg.sender).MARKET_ID()), and if the accumulated supplyAmount meets or exceeds minLoan, it supplies the full accumulated amount into Moolah on behalf of the configured vault and resets supplyAmount to zero. This batching behavior ensures that small interest flows do not create invalid or dust-sized vault positions.

For discounted interest repayment, brokers call transferLoan(amount) to receive loan tokens from the relayer. This function enforces that the relayer only transfers from the portion of its balance that is not reserved for vault supply by computing remainingLoan = balance - supplyAmount, and requiring amount <= remainingLoan. The relayer then transfers the requested loan tokens to the broker, allowing the broker to proceed with repayment settlement while preserving the integrity of the pending vault-supply buffer.

The relayer additionally provides a manager-controlled withdrawLoan(amount, receiver) function that allows the MANAGER to extract loan tokens that are not reserved in supplyAmount, subject to the same remainingLoan constraint, enabling administrative recovery of excess or stranded balances without impacting the next vault supply batch. The contract is upgradeable via UUPS, with upgrades authorized by DEFAULT_ADMIN_ROLE.

Appendix: Whitelisting Mechanism

Only addresses contained in the internal brokers set may call supplyToVault and transferLoan. Whitelisting is performed by addBroker(broker), which requires that the broker is not already present and that ICreditBrokerBase(broker).LOAN_TOKEN() equals the relayer's configured token. This creates a single-asset invariant per relayer instance: all whitelisted brokers must operate on the same loan token, preventing mixed-asset accounting for supplyAmount and preventing a broker from draining balances denominated for another asset.

Removal is performed via removeBroker(broker), immediately revoking access to relay functions, while historical balances remain in the relayer and continue to be governed by supplyAmount reservation rules.

Appendix: Reservation Accounting

The relayer tracks supplyAmount as the amount of loan token currently reserved for eventual supply into the Moolah vault. Each call to supplyToVault(amount) increases supplyAmount after transferring tokens in, then checks the relevant minLoan threshold for the calling broker's market. When supplyAmount >= minLoan, the relayer supplies exactly supplyAmount to Moolah and resets supplyAmount to zero, batching multiple broker interest transfers into fewer vault supplies.

The reservation design is enforced by computing remainingLoan = balance - supplyAmount when transferring loan tokens out of the relayer. Both transferLoan (broker) and withdrawLoan (manager) are constrained by amount <= remainingLoan, ensuring that funds earmarked for vault supply cannot be extracted and that the relayer cannot become undercollateralized relative to its pending vault obligation.

Privileged Functions

- addBroker
- removeBroker
- withdrawLoan

Core Invariants:

INV 1: Should only transfer tokens that are in excess of supplyAmount

INV 2: Only brokers should be apply to supply assets

INV 3: supplyToVault call should not revert in any proper flow

Issue_48	Lista tokens remain stuck
Severity	**High**
Description	Currently there is no way to withdraw LISTA tokens which have been accumulated via the interest repayment with LISTA, which means these tokens will just remain stuck.
Recommendations	Consider implementing a withdrawal function.
Comments / Resolution	Resolved

Issue_49	Repayments may revert due to truncation within vault supply
Severity	High
Description	When supplying to Moolah, it checks for minLoan and only executes the supply if indeed supplyAmount >= minLoan. If supplyAmount >= minLoan (but not too large), it calls supply with the asset amount (supplyAmount). It then calculates shares to be received via truncation and then finally checks that the asset amount for shares received (also rounded down) is above minLoan. This will revert in different scenarios, see example: - totalSupplyAssets = 1000 - totalSupplyShares = 100 - minLoan = 101 - supplyAssets = 105 - it passes minLoan check - calls moolah.supply - shares = floor(105 * 100 / 1000) - shares = 10 - convert shares to amount for min check - assets = 10 * 1000 / 100 - assets = 100 - assets is below minLoan -> revert -> revert whole control-flow of repay While it is expected that the current market is interest free, the Relayer can have multiple different brokers and since the market is derived from the msg.sender: *MOOLAH.supply(MOOLAH.idToMarketParams(ICreditBrokerBase(msg.sender).MARKET_ID()), _supplyToVault, 0, vault, "");* It is possible that the corresponding market has interest which results in totalSupplyAssets > totalSupplyShares and thus truncations.

Recommendations	Consider implementing a buffer which makes sure such truncation scenarios do not revert.
Comments / Resolution	Acknowledged. CreditCollateral markets will not have any interest and liquidations hence maintaining the share:assets ratio at 1

Issue_50	Violation of CEI
Severity	**Low**
Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes). supplyToVault increases totalSupply only after the external transfer
Recommendations	Consider following the CEI pattern.
Comments / Resolution	Resolved.

Issue_51	Different markets may have different minLoan return values
Severity	**Informational**
Description	Different markets may have different minLoanAmount values which means supplyAmount may be insufficient for one but sufficient for the other and this will create ambiguous situations which can be influenced by users.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_52	Possibly redundant minLoan check if vault holds shares
Severity	**Informational**
Description	During the supply logic of interest/penalty, execution only happens if the amount is above the minLoan. This is not actually needed if the vault already owns some shares.
Recommendations	Consider acknowledging this note.
Comments / Resolution	Acknowledged.

Issue_53	Relayer requires LOAN_TOKEN to be supplied manually
Severity	**Informational**
Description	During the repay with LISTA control-flow, it is expected that the relayer owns some LOAN_TOKEN which is then transferred to the CreditBroker contract. Currently, there is no such automatism for this which means it all relies on governance manually transferring the LOAN_TOKEN to this contract.
Recommendations	Consider keeping this in mind.
Comments / Resolution	Acknowledged.

Token

CreditToken

The CreditToken contract is a custom ERC20 implementation which uses an off-chain module to distribute credit tokens to eligible users via a merkle root mechanism.

Based on various different mechanisms, such as:

- KYC
- Social Media Info
- Github Info
- TX History
- ....

CreditTokens are being distributed to users via the different sync functions (bulkSyncCreditScores, syncCreditScore). It essentially stores for each user a CreditScore struct with id and score, whereas id corresponds to the current versionId of the merkle root (increments whenever a new merkle root is accepted) and score corresponds to the current eligible amount of credit tokens.

These minted tokens are then stored for each user within the userAmounts mapping. At any time it can happen that a new merkle root is set which then possibly mints or burns tokens from users based on the userScore and the userAmounts value. If userScore is larger than userAmounts it will mint tokens as that means the user is eligible for more credit tokens. Vice-versa it burns tokens whenever the user is not eligible anymore for credits.

Transfers are currently limited to the CreditBroker and to the Moolah contract as these credit tokens are only meant to be used in a closed system and nowhere outside of the system.

Appendix: Debt Accumulation

Since credit tokens are meant to be supplied into Moolah as collateral to borrow U (https://bscscan.com/address/0xcE24439F2D9C6a2289F741120FE202248B666666), it is expected for user's balance of the credit token to drop, because now the Moolah contract holds these tokens. If a new merkle root is set, it can happen that userScore < userAmount and then it naturally attempts to burn the delta to ensure that userScore = userAmount to meet a user's eligible amount of credit tokens.

However, if there is no balance to burn / insufficient balance to burn, it cannot meet this target and after the burn, userAmount > userScore, which means the user currently has a

debt. It is expected that users repay their debt within a short period of time and if that does not happen, they will simply be blacklisted from the credit system for the future.

Appendix: Two-Step Merkle Root Setting

The merkle root is used as a fundamental source of truth to determine the userScore for users which are eligible for credit tokens. Currently, the contract enforces a two-step mechanism for setting the merkle root which first goes through setPendingMerkleRoot, then through a waiting period (waitingPeriod) and then finally through acceptMerkleRoot, which sets the new merkle root and increments the versionId.

This process can only be conducted by the BOT address.

Privileged Functions

- transfer
- transferFrom
- setPendingMerkleRoot
- acceptMerkleRoot
- revokeMerkleRoot
- changeWaitingPeriod

Core Invariants:

INV 1: lastSetTime must be type(uint256).max if there is no pending root

INV 2: pendingMerkleRoot must be bytes32(0) if there is no pending root

INV 3: If there is a pending root, can never call setPendingMerkleRoot

INV 4: versionID must be incremented whenever acceptMerkleRoot is being called

INV 5: revokePendingMerkleRoot is only callable if there is a pending merkle root

INV 6: During syncCreditScore, a user's versionId must always match with the current versionId

INV 7: Should only be transferable by Moolah or CreditBroker

Issue_54	Token can be transferred to other addresses via new markets
Severity	**High**
Description	The Moolah contract allows for permissionless market creation if there are no operators. Such a setting opens an exploit vector where users can create a new market with the COLLATERAL_TOKEN. Since the Moolah contract has the TRANSFERER role, users can deposit into the new market on behalf of another address or withdraw to another recipient which then transfers the credit token to an address which is never entitled to hold it. Additional note resolution 1: During the initial market creation, it should be ensured that no users have credit tokens at the time between market creation and setProvider/setBroker calls, to prevent this problem. Alternatively it can also be ensured that market creation and setProvider/Broker is permissionless. A third alternative could be to disable transferrer roles on the CreditToken before the initial setup has been finalized.
Recommendations	Consider ensuring that no new market with the credit token can be created by normal users.
Comments / Resolution	Acknowledged.

Issue_55	Lack of debt payment incentive
Severity	**Medium**
Description	Whenever a user has transferred funds to the Moolah contract and then a userScore decrease happens, the userAmount is not sufficiently decreased because of a lack of token burns. This means a user will essentially enter a debt state. There is currently no incentive mechanism for users to repay their debt, more so, users can intentionally supply right before a userScore decrease to prevent burning of tokens.
Recommendations	Consider implementing a solution for this.
Comments / Resolution	Acknowledged.

Issue_56	Pausing is not enabled
Severity	**Medium**
Description	The contract implements the pausable pattern but fails to expose the pause/unpause functions which essentially makes it impossible to pause the contract in emergency situations.
Recommendations	Consider implementing pause/unpause for the MANAGER
Comments / Resolution	Resolved.

Issue_57	Two-step mechanism of merkleRoot setting makes it easy for users to foresee score decrease
Severity	**Low**
Description	The merkle root setting is a two-step process with a setting delay. Users can determine a pending setting to then execute a supply call just before the pending merkle root is set, potentially escaping a burn of their credit tokens.
Recommendations	Consider removing the two-step process.
Comments / Resolution	Acknowledged.

Issue_58	Third condition during _syncCreditScore is redundant
Severity	**Low**
Description	The third condition in _syncCreditScore assumes that userAmount > userScore via debtOf(). However, this state is already covered in the second condition, rendering the third condition unreachable.
Recommendations	Consider removing the third condition.
Comments / Resolution	Resolved.

Issue_59	Incorrect event emission within ScoreSynced event
Severity	**Low**
Description	The ScoreSynced event currently represents the updated score and versionId as the old ones, which makes the event emission incorrect.
Recommendations	Consider using the old score and versionID
Comments / Resolution	Resolved.

Issue_60	syncCreditScore can be called for address which is not existent in system
Severity	**Informational**
Description	The syncCreditScore function is currently callable for any address including those that are not part of the merkle system. While we could not identify any harm due to this because the default version and score will be zero, we are of the opinion that it is valuable to make a note for this behavior.
Recommendations	Consider acknowledging this behavior.
Comments / Resolution	Acknowledged.