Liquify

Borrowing And Lending Component Audit Report

Version 1.0

Conducted by:
Alin Barbatei (ABA) — Lead Silverologist

July 9, 2025

Table of Contents

1 Introduction

1.1 About The Auditors

ABA, or Alin Mihai Barbatei, is an established independent security researcher with deep expertise in blockchain security. With a background in traditional information security and competitive hacking, ABA has a proven track record of discovering hidden vulnerabilities. He has extensive experience in securing both EVM (Ethereum Virtual Machine) compatible blockchain projects and Bitcoin L2, Stacks projects.

Having conducted several solo and collaborative smart contract security reviews, ABA consistently strives to provide top-quality security auditing services. His dedication to the field is evident in the top-notch, high-quality, comprehensive smart contract auditing services he offers.

To learn more about his services, visit ABA's website abarbatei.xyz. You can also view his audit portfolio here. For audit bookings and security review inquiries, you can reach out to ABA on Telegram, Twitter (X) or WarpCast:

`https://t.me/abarbatei`

`https://x.com/abarbatei`

`https://warpcast.com/abarbatei.eth`

For this audit, Silverologist (@silverologist), a promising auditor, contributed their expertise to help secure the protocol.

1.2 About Liquify Borrowing And Lending Component

The Liquify Borrowing and Lending component is a new product in the Liquify Ventures ecosystem. Borrowers put up collateral and draft the terms of the loan themselves while the lenders provide the underlying loan amount through a fundraising mechanism.

The system does not use any external oracle, relying on market participants to correctly estimate their assets worth when accepting to participate in any existing loan.

1.3 Issues Risk Classification

The current report contains issues, or findings, that impact the protocol. Depending on the likelihood of the issue appearing and its impact (damage), an issue is in one of four risk categories or severities: *Critical*, *High*, *Medium*, *Low* or *Informational*.

The following table shows an overview of how likelihood and impact determines the severity of an issue.

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behavior that's not so critical.

Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

Informational findings

Informational findings encompass recommendations to enhance code style, operations alignment with industry best practices, gas optimizations, adherence to documentation, standards, and overall contract design.

Informational findings typically have minimal impact on code functionality or security risk.

Evaluating all vulnerability types, including informational ones, is crucial for a comprehensive security audit to ensure robustness and reliability.

2 Executive Summary

2.1 Overview

Project Name	Liquify Borrowing And Lending Component
Codebase	https://github.com/Liquify-labs/lending-and-borrowing
Operating platform	Arbitrum, Unichain, Polygon, Base, Optimism, Ethereum, Berachain, Sonic, Soneium, BSC
Programming language	Solidity
Initial commit	`6fbf379f3fdfcce544e6023271aaa4adb1eeb6f1`
Remediation commit	`7d9bdb142e1e1ce3733b791531041f2ff3f360ac`
Timeline	From 05.06.2025 to 08.06.2025 (4 days)
Audit methodology	Static analysis and manual review

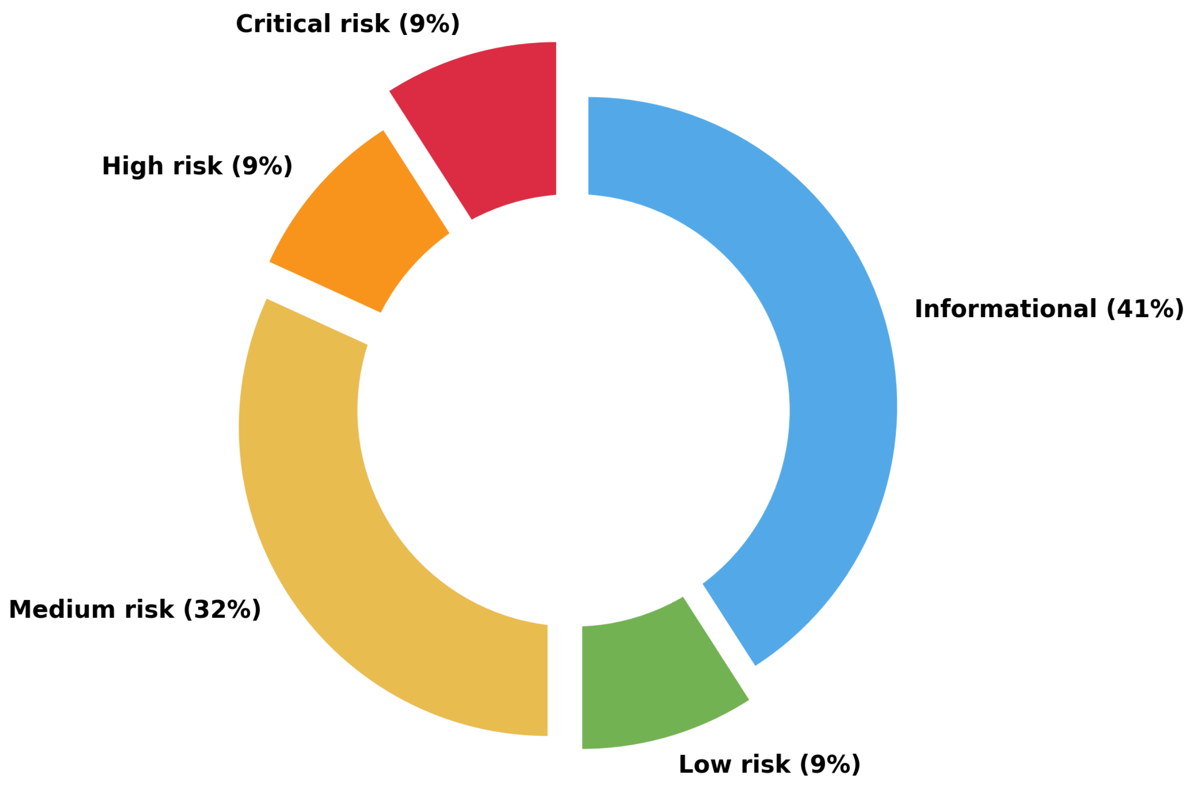
2.2 Audit Scope

Files and folders in scope

```
- contracts/LoanManager.sol
- contracts/LoanShare.sol
- contracts/interfaces/ILoanManager.sol
```

2.3 Summary of Findings

Severity	**Total Found**	**Resolved**	**Partially Resolved**	**Acknowledged**
Critical risk	2	2	0	0
High risk	2	2	0	0
Medium risk	7	6	0	1
Low risk	2	2	0	0
Informational	9	2	0	7



2.4 Findings & Resolutions

ID	Title	Severity	Status
C-01	Borrower can redeem collateral of unpaid loan	Critical	Resolved
C-02	Borrowers don’t receive their collateral back on payment	Critical	Resolved
H-01	Penalty fee retained on unfunded loans results in locked collateral	High	Resolved
H-02	Borrower cannot withdraw collateral after all lenders withdraw funding from unclaimed loan	High	Resolved
M-01	Unclaimed penalty parameter change leads to fund imbalance	Medium	Resolved
M-02	Protocol fee changes during loan lifecycle can cause fund loss	Medium	Resolved
M-03	Missing rejection fee allows griefing through repeated loan cancellations	Medium	Resolved

ID	Title	Severity	Status
M-04	Updating BORROWER_CLAIM_EXTENSION may lead to unexpected liquidation	Medium	Resolved
M-05	Partial lenders receive disproportionate collateral after loan default	Medium	Acknowledged
M-06	Penalty fee is overcharged for underfunded loans	Medium	Resolved
M-07	Pausing protocol leads to borrower-lender issues	Medium	Resolved
L-01	Unrepayable loans can be created	Low	Resolved
L-02	Owner gated functions should not be pausable	Low	Resolved
I-01	Full ERC20 deployment per loan leads to increased loan creation gas costs	Informational	Acknowledged
I-02	Undeclared error identifier	Informational	Resolved
I-03	Borrower may unfairly be charged unclaimed penalty fee due to sequencer downtime	Informational	Acknowledged
I-04	Protocol does not support rebase or fee-on-transfer tokens	Informational	Resolved
I-05	Off-by-one error in setProtocolFee	Informational	Acknowledged
I-06	Add security contact to contracts	Informational	Acknowledged
I-07	Missing ProtocolFeeUpdated event emission in constructor	Informational	Acknowledged
I-08	LoanState enum uses valid state as default	Informational	Acknowledged
I-09	Use a two-step ownership transfer routine	Informational	Acknowledged

3 Findings

3.1 Critical Severity Findings

[C-01] Borrower can redeem collateral of unpaid loan

Severity: ***Critical risk*** *(Resolved)* *[PoC]*

Context: ***LoanManager.sol***:*314*

Description

A loan can enter the `DEFAULTED` state in two different scenarios:

- Claimed but not repaid: The borrower claims the loan but fails to repay it by the repayment deadline. In this case, the collateral is meant to compensate lenders.
- Unclaimed past deadline: The borrower fails to claim or reject the loan by the `fundingDeadline + BORROWER_CLAIM_EXTENSION` deadline. In this scenario, the borrower pays a penalty, lenders withdraw their funds with a share of the collateral, and the borrower can redeem the remainder.

Both of these distinct scenarios currently share the same `DEFAULTED` state, leading to two major issues.

Firstly, borrowers who claim but don't repay their loans can still call `redeemExcessCollateral`, allowing them to recover collateral without repayment. This removes any incentive to repay the loan.

Secondly, the flow for unclaimed loans is broken. In the unclaimed loan scenario, the expected flow is:

1. Borrower creates a loan.
2. Lenders fund it.
3. `fundingDeadline + BORROWER_CLAIM_EXTENSION` is reached.
4. Lenders call `withdrawFunding` to retrieve their funding and some penalty collateral.
5. The loan is liquidated.
6. Borrower redeems remaining collateral after penalty.

However, the protocol does not enforce this sequence. The loan can be marked as liquidated (`DEFAULTED`) before lenders call `withdrawFunding`, which causes it to revert. As a result, lenders are forced to call `claimCollateral`, which exposes three problems:

- Locked USDC: The USDC deposited by lenders has not been claimed by the borrower (who missed the deadline), but lenders' shares are burned when they claim collateral. Since the borrower can't claim the USDC either, these funds become permanently locked in the contract.
- Collateral Double-Spending: If lenders call `claimCollateral` first, they receive a pro rata share of the loan's total collateral. However, the borrower can still withdraw the full collateral minus the penalty, effectively double spending the collateral at the expense of other loans using the same collateral token.

- Lender Losses: If the borrower withdraws first, they receive the collateral minus the penalty. Then, when lenders call `claimCollateral`, they only receive a share of that penalty instead of the full amount owed, without recovering their USDC. This results in significant losses for lenders.

Recommendation

Separate the current `DEFAULTED` state into two states:

- `DEFAULTED_NOT_REPAID` for loans that were claimed but not repaid
- `DEFAULTED_NOT_CLAIMED` for loans that were not claimed in due time

Then, update function access as follows:

- `withdrawFunding` should only be callable in the `DEFAULTED_NOT_CLAIMED` state
- `claimCollateral` should only be callable in the `DEFAULTED_NOT_REPAID` state
- `redeemExcessCollateral` should only be callable in the `DEFAULTED_NOT_CLAIMED` state

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

[C-02] Borrowers don’t receive their collateral back on payment

Severity: *Critical risk* *(Resolved)*

Context: ***LoanManager.sol***:*228-257*

Description

After a loan is repaid by calling the `LoanManager::repay` function, the borrower should receive his collateral back.

However, no collateral is sent back, leaving it blocked in the smart contract.

Note: issue was identified by the development team during the audit.

Recommendation

In the `repay` function, after the loan was repaid and funds transferred, also transfer the borrower collateral back to him.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

3.2 High Severity Findings

[H-01] Penalty fee retained on unfunded loans results in locked collateral

Severity: ***High risk*** *(Resolved) [PoC]*

Context: ***LoanManager.sol****:406*

Description

If a borrower creates a loan but fails to claim or reject it before the deadline, the loan can be marked as defaulted. In this case, the borrower can withdraw their remaining collateral using `redeemExcessCollateral`, which deducts an unclaimed penalty fee from the total collateral.

This penalty is deducted even if the loan is entirely unfunded. This leads to a scenario where penalty collateral is retained without any eligible party to claim it, resulting in permanently locked tokens.

Consider the following scenario:

- `alice` creates a loan
- no lenders fund the loan
- `alice` misses the rejection deadline
- the loan is liquidated
- she calls `redeemExcessCollateral`
- the protocol deducts a penalty fee, despite no lender participation
- the deducted collateral fee remains locked in the contract with no beneficiary

Recommendation

Update `redeemExcessCollateral` to either apply the penalty fee only when the loan has received funding, or to redirect the penalty fee to the protocol owner when no funds were raised for the loan.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

[H-02] Borrower cannot withdraw collateral after all lenders withdraw funding from unclaimed loan

Severity: ***High risk*** *(Resolved) [PoC]*

Context: ***LoanManager.sol****:386*

Description

If a borrower fails to claim or reject a loan within the allowed window, lenders can begin withdrawing their funds via the `withdrawFunding` function.

Once the last lender withdraws their USDC, the loan state is set to `Cancelled`:

```
        // if 'everyones now burned their share tokens, cancel the loan
        if (share.totalSupply() == 0) {
            L.state = LoanState.Cancelled;
            emit LoanCancelled(loanId);
        }
```

However, this prevents the borrower from recovering their collateral since:

- `redeemExcessCollateral` is only available when the loan is in a `Defaulted` state.
- Liquidation is not possible when the loan is marked as `Cancelled`.

As a result, the collateral remains permanently locked in the `LoanManager`, resulting in a loss for the borrower.

Recommendation

Do not set the loan state to `Cancelled` when all lenders withdraw funds from an unclaimed loan.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

3.3 Medium Severity Findings

[M-01] Unclaimed penalty parameter change leads to fund imbalance

Severity: *Medium risk* (*Resolved*)

Context: *LoanManager.sol*:404,365,65

Description

In the `LoanManager` contract, if a borrower fails to claim a loan during the claim extension period, lenders are allowed to withdraw their initial USDC contributions along with a bonus, calculated using the `unclaimedPenaltyBps` parameter.

When the borrower later retrieves their excess collateral, a portion equal to `unclaimedPenaltyBps` is withheld to account for the lenders' bonus.

However, the `LoanManager` owner can update the `unclaimedPenaltyBps` value at any time using the `setUnclaimedPenaltyBps` function. If this value is changed in the middle of the lenders and borrower withdrawing the loan artifacts, the amount deducted from the borrower's collateral may not match the amount distributed to lenders.

This mismatch can lead to:

- Locked funds, if less is distributed to lenders than deducted from the borrower.
- Insufficient tokens to settle this or other loans, if more is distributed to lenders than was actually deducted from the borrower's collateral

Recommendation

Store the value of `unclaimedPenaltyBps` in the `Loan` struct at the time of loan creation, and use the struct value instead of the global one.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

[M-02] Protocol fee changes during loan lifecycle can cause fund loss

Severity: *Medium risk* (*Resolved*)

Context: *LoanManager.sol*:246

Description

The protocol fee is charged when a borrower repays a loan using the `repay` function.

This fee percentage can be modified at any time by the protocol owner through the `setProtocolFee` function.

If the fee is updated after a loan is created but before it is repaid, the borrower may end up paying a different fee than originally expected.

If the fee is increased, the borrower pays more than agreed, resulting in a loss for them.

If the fee is decreased, the protocol collects less than expected, leading to a loss for the protocol.

Recommendation

Store the current `protocolFeeBps` in the `Loan` struct at the time the loan is created. Use this stored value for fee calculations instead of referencing the global variable.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

[M-03] Missing rejection fee allows griefing through repeated loan cancellations

Severity: *Medium risk* *(Resolved)*

Context: ***LoanManager.sol***:*380*

Description

In the `LoanManager` contract, lenders can fund loans initiated by borrowers. Once the fundraising deadline has passed, the borrower is given a window of time to either claim, or reject the loan if the funding target has not been reached.

If the borrower chooses to reject the loan, they are not required to pay any rejection fee. As a result, lenders can only retrieve their initial contributions without any interest.

This opens the door to a griefing attack: an attacker can create numerous loans with unrealistic targets and intentionally reject them after fundraising ends. This traps lender funds for the duration of the fundraising period plus the borrower's claim window, without yielding any return.

Repeating this behavior can erode lender confidence in the protocol, as it exposes them to the risk of capital lockup without compensation.

Recommendation

Introduce a loan rejection fee calculated in basis points. When a borrower rejects a loan, remove this fee from the returned collateral. Distribute the fee proportionally to lenders when they withdraw funds from the rejected loan, or send it to the protocol owner.

Ensure that the rejection fee is less than the penalty for unclaimed loans, otherwise borrowers are not incentivized to reject the loan in due time.

Resolution: Resolved. A fix was implemented in **#c7b5733**.

ABA: the team implemented a rejection fee which is taken only if at least 10% of the loan amount was raised.

[M-04] Updating BORROWER_CLAIM_EXTENSION may lead to unexpected liquidation

Severity: *Medium risk* *(Resolved)*

Context: ***LoanManager.sol****:468*

Description

`BORROWER_CLAIM_EXTENSION` defines the period a borrower has to either accept or reject a loan offer without incurring penalties.

If this period is reduced after a loan has already been issued, the borrower might face unexpected liquidation due to having less time than initially expected.

Conversely, increasing this period post-loan creation can be unfair to lenders, as their funds may remain locked for longer than originally agreed upon if the loan is ultimately declined.

Recommendation

Store the current `BORROWER_CLAIM_EXTENSION` in the `Loan` struct at the time the loan is created. Use this stored value for time limit calculations instead of referencing the global variable.

Additionally, rename `BORROWER_CLAIM_EXTENSION` to `borrowerClaimExtension`, to comply with the Solidity style guide.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

[M-05] Partial lenders receive disproportionate collateral after loan default

Severity: *Medium risk* *(Acknowledged)* *[PoC]*

Context: ***LoanManager.sol****:325*

Description

When a loan is marked as defaulted, lenders can invoke `claimCollateral` to redeem a share of the loan's collateral based on their contribution.

However, the current implementation does not account for underfunded loans. Regardless of how much of the loan's target amount was actually funded, the entire collateral is distributed among the participating lenders. This means that even if only a small portion of the loan was funded e.g. a few wei, lenders can still collectively claim 100% of the collateral, leading to disproportionate and unjustified returns.

Recommendation

Update the `claimCollateral` logic to ensure that the distributed collateral is proportional to the amount of the loan actually funded versus the original loan target.

Resolution: Acknowledged by the team.

[M-06] Penalty fee is overcharged for underfunded loans

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** *LoanManager.sol:404*

Description

If a borrower creates a loan but fails to claim or reject it before the deadline, the loan can be marked as defaulted. In this case, the borrower can withdraw their remaining collateral using `redeemExcessCollateral`, which deducts an unclaimed penalty fee from the total collateral.

Currently, the penalty is calculated on the entire collateral amount, regardless of how much of the loan was actually funded. This results in:

- Excessive returns for lenders when only a small portion of the loan was funded.
- Unfair losses for the borrower, who pays a penalty on collateral that was intended to back the unfunded part of the loan.

Recommendation

Update `redeemExcessCollateral` to apply the penalty only to the portion of collateral corresponding to the amount of the loan that was actually funded.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

[M-07] Pausing protocol leads to borrower-lender issues

> **Severity:** *Medium risk* *(Resolved)*
>
> **Context:** Global

Description

The protocol owner can pause and unpause loan related actions (excluding withdrawals) at any time.

This creates a risk where borrowers may be unable to act if the protocol is paused during their allowed time window. As a result, borrowers will incur unfair penalties or lose access to their collateral through no fault of their own.

Recommendation

One potential workaround which comes with severe overhead, is to implement a grace period and track pauses:

- Apply a grace period (e.g. 1 hour) after unpausing during which loans that were not liquidatable before the pause cannot be liquidated.
- Record each pause's start and end time.
- Set a cap for the number of tracked pauses (e.g. 100 most recent pauses) to avoid loan actions reverting due to out of gas errors.

- Whenever a loan deadline is checked, offset it to the current timestamp plus the grace period if a pause overlapping with the action window prevented the borrower from taking action.
- Handle edge cases, such as:
  - A pause that covers both the claim and the repayment time windows.
  - A pause that ends just before a loan's deadline, leaving an unreasonably short time for action.

This approach protects borrowers but also negatively impacts lenders, as their returns are not adjusted to reflect the extended duration their funds remain locked in the loan.

An alternative solution is to allow all loan actions to be performed even while the protocol is paused, while disallowing new loan creation. This would minimize disruptions for existing loans, though it weakens the effectiveness of the pause mechanism.

Normal borrowing and lending markets use a simple grace period where liquidations are not allowed after unpause, however that approach brings the mentioned overhead in this situation, thus we recommend to either:

- adopt the alternative solution to only disallow loan creation during pauses or
- clearly document the borrower risk associated with pauses and acknowledge this issue.

Resolution: Resolved. A fix was implemented in **#c7b5733**.

ABA: the team modified so that only creating a loan or funding an existing one can be paused. While this may result in loans that are in the fundraising stage being rejected, it is an acceptable compromise when wanting to block any inflows into the project.

3.4 Low Severity Findings

[L-01] Unrepayable loans can be created

Severity: *Low risk* (*Resolved*)

Context: ***LoanManager.sol****:90*

Description

When a loan is created, the borrower can set any non-zero value for both the funding duration and repayment duration. The claim extension duration is implicitly set to 2 days.

Consider the following scenario:

- `alice` creates a loan with a 1-day funding duration and a 1-day repayment duration.
- `alice` claims the loan 2.5 days after its creation, which is allowed, as she's within 2 days of the funding deadline.
- `alice` is unable to repay the loan even immediately, since the repayment deadline has already passed.

This leads to unrepayable loans being created and claimed.

Recommendation

Implement a minimum repayment time that is larger than `BORROWER_CLAIM_EXTENSION`.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

[L-02] Owner gated functions should not be pausable

Severity: *Low risk* (*Resolved*)

Context: ***LoanManager.sol****:56,67,470*

Description

The `LoanManager` has several operations that can be paused.

Out of these, the functions `setBorrowerClaimExtension`, `setProtocolFee` and `setUnclaimedPenaltyBps` and functions that can only be called by the owner of the contract, but they are also blocked when the pause state is initialized.

As the owner is also the only entity allowed to pause the contract, gating any other only-owner functions is redundant and may increase logistics difficulty in case the protocol wishes to slightly pause until it reconfigures itself.

Recommendation

Remove the `whenNotPaused` modifier from any `onlyOwner` modifier gated functions. Specifically for the `setBorrowerClaimExtension`, `setProtocolFee` and `setUnclaimedPenaltyBps` functions.

Resolution: Resolved, the recommended fix was implemented in **#c7b5733**.

3.5 Informational Findings

[I-01] Full ERC20 deployment per loan leads to increased loan creation gas costs

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** ***LoanManager.sol***:*108*

Description

Each time a loan is created, the contract deploys a new instance of the `LoanShare` ERC20 contract to represent lender shares.

Since the logic and bytecode of every `LoanShare` contract are identical, repeatedly deploying full contracts is unnecessarily expensive. This leads to increased gas consumption and higher loan creation costs.

Recommendation

Use the OpenZeppelin Clones library to deploy minimal proxy contracts for `LoanShare`.

Resolution: Acknowledged by the team.

[I-02] Undeclared error identifier

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** ***LoanManager.sol***:*44*

Description The `LoanManager` constructor uses the `InvalidUSDCDecimals` error.

However, this error is not declared, blocking project compilation.

Recommendation

Declare the `InvalidUSDCDecimals` error in the `ILoanManager` interface.

Resolution: Resolved, the recommended fix was implemented in **#5fe55a2**.

[I-03] Borrower may unfairly be charged unclaimed penalty fee due to sequencer downtime

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** ***LoanManager.sol***:*404*

Description

If a borrower fails to claim or reject a loan within the designated time frame, they are required to pay a penalty fee to the lenders, as defined by `unclaimedPenaltyBps`.

However, if the borrower is unable to take action due to L2 sequencer downtime, they may miss the claim/rejection window. If the downtime lasts long enough to cause the window to expire, the borrower will be unfairly forced to pay the penalty fee.

Recommendation

A possible solution would be to implement a grace period for borrowers who cannot claim or reject their loans due to sequencer downtime. This can be done utilizing the specific APIs for specific blockchains.

However, the extra overhead may not provide enough value, as such a compromise would be to clearly document and acknowledge this issue.

Resolution: Acknowledged by the team.

[I-04] Protocol does not support rebase or fee-on-transfer tokens

> **Severity:** *Informational* *(Resolved)*
>
> **Context:** Global

Description

The `LoanManager` contract does not account for tokens with non-standard behaviors, such as rebasing or fee-on-transfer mechanics.

While the protocol intends to support only standard ERC-20 tokens with 18 decimals, this restriction is not enforced. As a result, borrowers can supply incompatible tokens as collateral, which can lead to loss of funds or locked assets.

Recommendation

A possible solution is to implement a whitelist of approved collateral tokens. This would restrict collateral to tokens vetted by the protocol, minimizing the risk of unexpected behavior.

Alternatively, clearly document this restriction and acknowledge this issue.

Resolution: Resolved. A fix was implemented in **#c7b5733**.

ABA: team decided to reject any token which has a fee on transfer. This check is done when creating the loan .

[I-05] Off-by-one error in setProtocolFee

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** ***LoanManager.sol****:54*

Description

There is a discrepancy between the natspec comment and the actual implementation of the `setProtocolFee` function:

- The `@dev` comment states that the maximum allowed fee is 10.00% (inclusive).
- The `@param` comment and the function logic enforce a maximum of less than 10.00% (exclusive).

This mismatch may cause confusion on what the actual maximum fee is.

Note, the same behavior is also observed in the constructor when the initial fee is set.

Recommendation

Align the documentation with the implementation by either:

- Updating the `@dev` comment to reflect the exclusive 10% cap, or
- Updating the `@param` comment and the implementation to allow a fee of 10.00%

Resolution: Acknowledged by the team.

[I-06] Add security contact to contracts

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** *LoanManager.sol*

Description

In case a whitehat identifies an issue with the on-chain contracts of the protocol, to more easily be able to contact the team, have a security contact added to the contracts.

Recommendation

Add the email address for the team security contact either as a plain comment or as a custom:security-contact natspec tag to the contracts.

Resolution: Acknowledged by the team.

[I-07] Missing ProtocolFeeUpdated event emission in constructor

> **Severity:** *Informational* *(Acknowledged)*
>
> **Context:** *LoanManager.sol:50*

Description

The `LoanManager` constructor sets the initial value of `protocolFeeBps`, but does not emit the corresponding `ProtocolFeeUpdated` event. This results in a missing event record of the initial fee configuration.

Recommendation

Emit the `ProtocolFeeUpdated` event within the `LoanManager` constructor.

Resolution: Acknowledged by the team.

[I-08] LoanState enum uses valid state as default

> **Severity:** ***Informational*** *(Acknowledged)*
>
> **Context:** ***ILoanManager.sol****:22*

Description

The `LoanState` enum currently uses `Fundraising` as its default (0) value. However, `Fundraising` is a valid state for active loans, which creates ambiguity when distinguishing between nonexistent and initialized loans.

Following best practices, the enum should reserve its first element as a clearly uninitialized state, e.g. `NULL`. This would simplify existence checks and improve event clarity in functions like `fund` and `withdrawFunding`, where operations on non-existent loans currently cause confusing errors to be raised.

Recommendation

Add a `NULL` or equivalent state as the first value in the `LoanState` enum to represent nonexistent loans. Then, remove the redundant loan existence check from `liquidate`:

```diff
-    if (loanId >= nextLoanId) revert WrongState();
```

Resolution: Acknowledged by the team.

[I-09] Use a two-step ownership transfer routine

> **Severity:** ***Informational*** *(Acknowledged)*
>
> **Context:** ***LoanManager.sol***

Description

When transferring ownership of a contract, mistakes that transfer the ownership to an unwarned address can be avoided by using a two-step transfer routine.

In the first step a new owner is proposed and in the second step the new owner must accept the ownership.

Recommendation

Use OpenZeppelin's 2-step ownership transfer contract in `LoanManager`.

Resolution: Acknowledged by the team.

4 Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts the consultant to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. The consultant’s position is that each company and individual are responsible for their own due diligence and continuous security. The consultant’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology that is analyzed.

The assessment services provided by the consultant is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. Furthermore, because a single assessment can never be considered comprehensive, multiple independent assessments paired with a bug bounty program are always recommend.

For each finding, the consultant provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved, but they may not be tested or functional code. These recommendations are not exhaustive, and the clients are encouraged to consider them as a starting point for further discussion.

The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties. Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, the consultant does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

The consultant retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. The consultant is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. The consultant is furthermore allowed to claim bug bounties from third-parties while doing so.