BAIL security

ALTO
Lending Market

FINAL REPORT

October ‘2025

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	ALTO - Lending Market - Audit Report
Website	altofoundation.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/altomoney/v1/tree/c965a0a49000824f1fa93e5007c06e25a826bdfa
Resolution 1	https://github.com/altomoney/v1/commit/a35f248251c74361bad637e1e9f35a65ba81942f

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no changes made)	Failed resolution	Open
High	3	3				
Medium	7	1	1	4		1
Low	9	1		8		
Informational	12	2		9		1
Governance	2			2		
Total	33	7	1	23		2

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

Security Advice

The present assessment acknowledges a set of critical boundary conditions and novel design choices that directly shape the interpretive scope of this audit. Each identified subsystem: oracle integration, destination-driven liquidation, router delegation, and surrounding authorization logic carries inherent risk dimensions that extend beyond the determinism of the verified code snapshot.

The following advisory serves to delineate the technical perimeter of the audit and provide a framework for post-deployment vigilance.

Oracle Integration

The oracle contract and its related pricing mechanisms are explicitly *out of scope* for this audit. It is assumed that decimal normalization and denomination handling are implemented correctly and that the oracle consistently returns prices in the intended base denomination.

Any deviation in these assumptions, such as incorrect scaling, stale pricing, or inconsistent base/quote mapping can propagate into severe accounting drift across the lending subsystem, potentially invalidating several invariant expectations established within the reviewed code.

The audit findings must therefore be interpreted under the condition that the oracle operates in a mathematically and temporally consistent manner.

Destination-Driven Liquidation Framework

The liquidation design represents a novel, destination-driven mechanism where the liquidation extent is algorithmically determined rather than supplied as a user input. While the logic adheres to formal mathematical invariants, its emergent behavior under adversarial boundary states remains only partially predictable.

BailSec formally categorizes this as a *non-deterministic risk surface*, where fuzzing and staged simulation become mandatory pre-deployment exercises.

Priority liquidators, once appointed, must be considered part of the trusted perimeter, as they effectively form the last control layer against recursive self-liquidation or state collapse in a bad-debt scenario.

It has to be noted that it might be a good design decision to keep the priority liquidation mechanism even in case of heavily undercollateralized positions (keyword: disablePriorityLiquidationAbovePositionLtv)

Any deployment without this trusted-party model materially increases systemic exposure.

Router and Delegated Execution Surface

Any potentially router makes use of delegated “onBehalf” actions routed through the AuthUpgradeable library. This pattern introduces composite control-flows where privileged calls may originate from an operator context rather than an end user.

In a production environment, the router will likely hold broad operator permissions, expanding the potential attack surface to include delegated execution misrouting and cross-context reentrancy. These scenarios were not covered within this audit round and must be formally acknowledged as residual risk until the router and AuthUpgradeable layers undergo a dedicated, isolation-focused review.

Scope Boundaries and Assumption Integrity

BailSec assessments are strictly code-referential: the audit conclusions are derived from the verified commit hash, not from documentation, implementation notes, or off-chain assumptions.

Certain issues may appear theoretical within the intended configuration but can manifest as critical if deployment parameters, token decimals, or reference integrations deviate from the verified environment.

In summary, the architectural composition of this protocol involves multiple trust edges whose correctness cannot be established within a static source-code audit. For that reason, the client and all deploying entities must regard this section as a **deployment-critical advisory**, mandating dedicated fuzzing, simulation, and post-deployment observability of the oracle, liquidation, and router layers to sustain the security guarantees implied herein.

AltoBaseMarket

The AltoBaseMarket is a lending contract which allows users to deposit the collateralToken and borrow the borrowToken, as long as the individual LTV stays below maxLTV.

A position can get liquidated as soon as the individualLTV >= maxLiquidationLTV and it is expected that maxLiquidationLTV > maxLTV in an effort to implement a buffer between how much can be borrowed and when a position can be liquidated.

The contract implements the usual interest rate pattern where borrowers are charged interest based on their borrowed amount and the time passed. A plug-in IRM is used which abstracts away the logic of calculating the rate.

The borrowToken is expected to be supplied via a separate addSupply function and it implements an ERC4626-like mechanism where the interest is simply accrued in the exchange rate, with the expectation that suppliers receive a constant interest rate stream. However, suppliers are at the same time subject to the risk of bad-debt as any bad-debt is offset to the suppliers.

The borrow accounting is also using an ERC4626-like system where users are assigned “borrow shares” which increase in value (debt) with each interest rate accrual.

The contract implements a novel liquidation mechanism which will be described in an appendix.

Appendix: Liquidation Flow

Before we start with the liquidation mechanics, we need to elaborate the idea behind liquidation variables:

- maxLTV: This is the LTV up to which users can borrow
- maxLiquidationLTV: This is the LTV when a liquidation starts
- LTVForCompleteLiquidation: This is the LTV when a position must be fully liquidated
- minPenaltyPercentage: This is the minimum liquidation percentage that a position must experience whenever it just reached the maxLTV

Destination-Driven Liquidation

Alto implemented a novel liquidation system which differs from the traditional liquidation mechanisms. Traditional liquidation mechanisms allow the liquidator to provide a specific amount to repay and then seize the corresponding collateral (including any fees/incentives).

Contrary to that, Alto developed a new mechanism which does not allow the user to specify a desired repayment amount but instead applies an algorithm that calculates how much percentage of the position should get liquidated, which is based on the value of the current LTV in relation to the maxLiquidationLTV and LTVForCompleteLiquidation:

> liquidationPercentage = (currentLTV - maxLiquidationLTV) * 1e18 / (LTVForCompleteLiquidation - maxLiquidationLTV)

This liquidationPercentage is then used to calculate the newLTV, which is simply "liquidationPercentage" percent below the maxLTV:

> newLTV = maxLTV * (1e18 - liquidationPercentage) / 1e18

Based on the newLTV, it is now calculated how much borrowToken is required to be repaid and how much collateralToken is seized:

> collateralToSellInBorrowAssets = (debt - (collateral * newLTV / 1e18)) * 1e18 / (1e18 - newLTV)

Now simply convert it into the collateral denomination and receive how much collateral to seize (plus the liquidation fee)

> totalCollateralToTake = (collateralToSellInBorrowAssets * oraclePrice / 1e36) + liquidatorBonusFee

Liquidation Percentage

The liquidationPercentage value is determined using the following three values:

- currentLTV
- maxLiquidationLTV
- LTVForCompleteLiquidation

Essentially, it derives the percentage value on what linear scale the currentLTV value is between [maxLiquidationLTV; LTVForCompleteLiquidation):

> liquidationPercentage = (currentLTV - maxLiquidationLTV) * 1e18 / (LTVForCompleteLiquidation - maxLiquidationLTV)

It is then used to determine the target LTV post-liquidation.

Target LTV

Each liquidation produces a specific newLTV which is the target LTV post liquidation for a position. This value is trivially determined by reducing the maxLTV based on the liquidationPercentage:

> newLTV = maxLTV * (1e18 - liquidationPercentage) / 1e18

It is then applied to calculate how much collateral must be seized and how much debt must be repaid to meet the newLTV.

Collateral to seize / Debt to repay

Based on newLTV/debt/collateral after fee, the actual required collateral seize amount/debt repayment is calculated. These values are equivalent but normalized to their corresponding denomination. If for example 1 WETH collateral is seized and the debt is denominated in USDT, it will require 4000 USDT to be repaid.

> collateralToSellInBorrowAssets = (debt - (collateral * newLTV / 1e18)) * 1e18 / (1e18 - newLTV)

This formula basically calculates: "**How much do i need to decrease DEBT and COLLATERAL (after fee) to reach the newLTV**?"

It is derived as follows:

We need to calculate how much the VALUE must be in an effort to reach the newLTV:

> (DEBT – VALUE) / (COLLATERAL – VALUE) = newLTV

Equivalent forming (solve to VALUE):

> DEBT – VALUE = newLTV × (COLLATERAL – VALUE)

Equivalent forming (solve to VALUE):

> DEBT – VALUE = (newLTV × COLLATERAL) – (newLTV * VALUE)

Equivalent forming (solve to VALUE):

> –VALUE + newLTV × VALUE = newLTV × COLLATERAL – DEBT

Equivalent forming (solve to VALUE):

> VALUE × (–1 + newLTV) = newLTV × COLLATERAL – DEBT

Equivalent forming (solve to VALUE):

> VALUE × (1 – newLTV) = DEBT – newLTV × COLLATERAL

Equivalent forming (solve to VALUE):

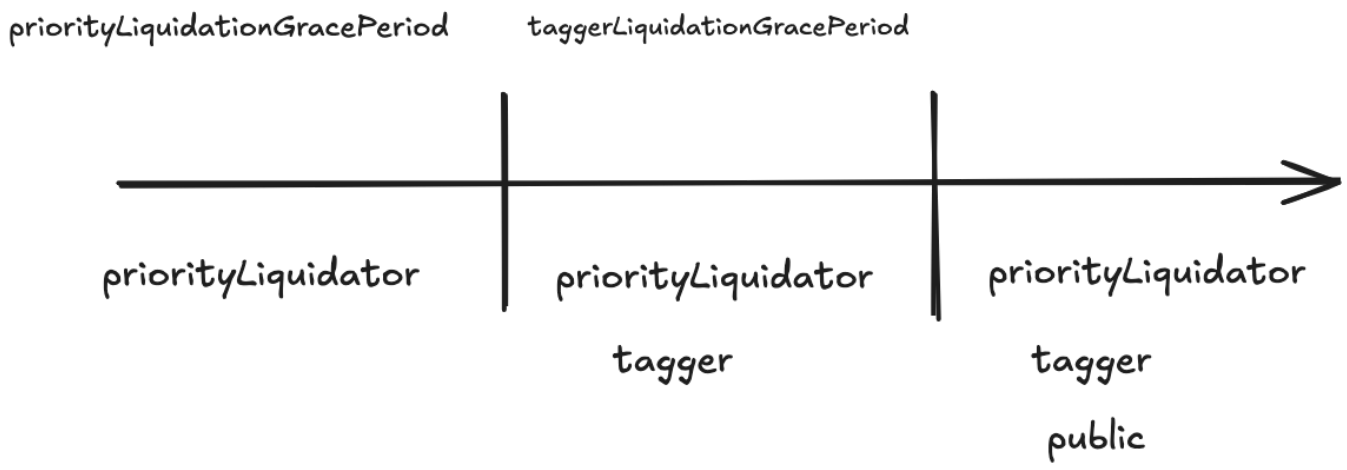
> VALUE = (DEBT – newLTV × COLLATERAL) / (1 – newLTV)

Tagging Mechanism

The codebase implements a tagging mechanism which allows to mark a position as liquidatablePosition[user] which is tied to the tagging time and the liquidator.

This liquidator can then liquidate the position within the priority liquidation mode, as soon as the priorityLiquidationGracePeriod has passed. Any other address can only liquidate after priorityLiquidationGracePeriod + taggerLiquidationGracePeriod.

Any priority liquidator can liquidate a position without being tagged and without waiting period.



Bad Debt Application

Whenever a position is liquidated which has a LTV > 1e18 or LTV near 1e18 (plus fees), it will happen that the repayment amount which is specified to meet newLTV = 0, is larger than the actual debt. This can be shown with a trivial example without any fee:

- price = 1
- COLLATERAL = 100e18
- DEBT = 105e18
- liquidationPercentage = 1e18
- newLTV = 0
- collateralToSellInBorrowAssets = 105e18
    - clamping to 100e18 due to exceeding COLLATERAL
- COLLATERAL is reduced to zero
- 100e18 DEBT is repaid
- A leftover debt with 5e18 is offset as bad debt

If a fee is incorporated, it will simply clamp collateralToSellInBorrowAssets even earlier because the COLLATERAL - FEE is used for the clamping purpose. Bad debt will be even larger.

Appendix: Hook Implementation

The contract implements hooks within the borrow and removeCollateral function which allows for leveraging and deleveraging positions. These features are described in the AltoLeverage contract.

Privileged Functions

- setIRM
- setOracle
- setMaxLTV
- setLiquidationConfiguration
- setFeeRecipient
- setAuthorizedCallback
- setPauser

Core Invariants:

INV 1: If a market is paused, interest should not be updated

INV 2: tagLiquidatablePosition is only possible if priorityLiquidation is enabled

INV 3: tagLiquidatablePosition is only allowed if liquidator is non-existent in priorityLiquidators[address] mapping

INV 4: Value of borrowShares must be constantly increasing

INV 5: Determination of collateralValue must always round down

INV 6: Determination of maxBorrowValue must always round down

INV 7: maxLTV must always be lower than maxLiquidationLTV

INV 8: _accrueInterest must increase totalSupply.assets and totalBorrow.assets similarly

INV 9: Liquidation is only allowed if a position is not considered as solvent

INV 10: priorityLiquidators do not receive a bonusFee (baseFee only)

INV 11: iquidationPercentage must be based on currentLTV progress between maxLiquidationLTV and LTVForCompleteLiquidation

INV 12: The oracle is responsible for normalizing decimals and convert the denomination to the borrowToken denomination

INV 13: At all times, totalSupply.assets >= totalBorrow.assets

INV 14: addCollateral must decrease LTV

INV 15: removeCollateral must increase LTV

INV 16: onBehalf must be solvent at the end of removeCollateral

INV 17: borrow must increase LTV

INV 18: onBehalf must be solvent at the end of borrow

INV 19: A user can never borrow more than maxLTV

INV 20: A user can only be liquidated once maxLiquidationLTV has been reached

INV 21: During liquidation with priority, if caller is tagger and not priorityLiquidator, must wait for priorityLiquidationGracePeriod to pass

INV 22: During liquidation with priority, if caller is not tagger and not priorityLiquidator, must wait for priorityLiquidationGracePeriod + taggerLiquidationGracePeriod to pass

INV 23: During liquidation with priority, can only do priority liquidation if currentLTV <= disablePriorityLiquidationAbovePositionLtv

INV 24: The liquidator must always receive: [equivalentCollateralOfRepayAmount + liquidatorBonus - protocolFee]

INV 25: After each liquidation, a user's position must be at least minPenaltyPercentage below maxLTV

INV 26: collateralToSellInBorrowAssets must not be larger than remainingCollateralAfterFeeInBorrowAssets and must not be larger than positionBorrowedAssets

INV 27: Bad debt must be applied if the repaid debt is equal to the collateral value but doesnt fully offset all debt

INV 28: Whenever a position is liquidated, the minimum LTV decrease must be by minPenaltyPercentage

INV 29: The higher the liquidationPercentage, the lower newLTV (down to zero for 100%)

INV 30: Whenever the market is setting to paused, it must accrue interest up to block.timestamp

INV 31: Whenever the market is setting to unpaused, it must set lastUpdate to block.timestamp

Issue_01	Equality possibility of maxLTV and maxLiquidationLTV raises serious self-liquidation concerns
Severity	**Governance**
Description	The liquidation mechanism offsets any potential bad-debt to suppliers. If there is any potential attack-vector which allows an attacker to artificially create a position with bad-debt which is self-liquidatable, this opens the door to stealing all funds from the contract, even if the bad-debt is only minimal, a repetitive execution of such a vulnerability allows for stealing all existing borrow tokens. The fact that the maxLTV can be set equal to maxLiquidationLTV increases the likelihood of such a scenario, as it essentially opens the door to self-liquidations: *if (_liquidationConfiguration.maxLiquidationLtv < _maxLtv) {* *revert AltoBaseMarketInvalidInput();* *}*
Recommendations	Consider not allowing maxLTV to be near or equal maxLiquidationLTV.
Comments / Resolution	Acknowledged. The Alto team is aware of such a risk and will always set a reasonable buffer.

Issue_02	Governance Issue: Full governance control
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. - Proxy pattern - Change of IRM and corresponding storage manipulation - Change of oracle - governanceLiquidate The function governanceLiquidate was introduced to liquidate positions that its collateral rounds down to zero so it can not be normally liquidated. If this function is used for normal liquidatable positions, no repayment will happen, it will offset the entire debt to the suppliers and take the whole collateral.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged

Issue_03	Reentrancy path via swapper can be exploited to offset bad-debt to suppliers
Severity	**High**
Description	Currently, the liquidation mechanism offsets bad debt to suppliers if the position has insufficient collateral to pay out to the liquidator to cover the repay amount. In this specific scenario, the repayment amount is simply capped with the current existing collateral. This means the position will end up with a state where: - collateral = 0 - debt != 0 In this state, the debt is simply offset to users by decreasing totalSupply.assets. This mechanism in itself is not a problem as long as an attacker is not able to force his own position into bad-debt and weaponize the offset. However, historically speaking, codebases were often vulnerable to such a situation in highly sophisticated edge-cases. Alto implements a leverage mechanism which simply allows users to leverage up their positions via the borrow function, selling borrowed funds into the collateralToken and increasing the collateral to result in a healthy position. Likewise with deleverage, users can use the removeCollateral function, swapping the collateral to the borrowToken and repaying their debt. All of this can happen in one TX due to the built-around system and thus users are not required to do manual looping. This mechanism exposes a preliminary invalid state where the position is heavily undercollateralized due to the withdrawal of collateral or the borrowing of funds. While a safety check at the end of the function has been implemented, this safety check is not sufficient. The attack is essentially enabled due to the fact that an external

	swapper is being used while the user can provide arbitrary swapData, including such data which allows for swapping through a pool with a malicious/self deployed token to hijack the control-flow. Essentially, an attacker can do the following steps to steal funds: a) Deposit some collateral and borrow funds b) Call the removeCollateral function and withdraw collateral while at the same time mimicking a deleverage action c) During this deleverage action, the attacker hijacks the control-flow while the position has a LTV > 1e18 and reenters into the liquidate function d) For this attack, the position can be in many different states but the easiest reproduction is a state where collateral = 0 and debt != 0, which is trivially reached during b) e) The liquidate function then simply clamps the actual repay amount to zero due to the collateral being zero and offsets all debt to the suppliers. The position is fully liquidated. f) The user now deposits assets again and borrows a small amount. This step is only to make the subsequent repay which is required for deleverage not revert, as the onRemoveCollateral control-flow requires for repay to be executed (even 1 wei execution is sufficient to prevent revert) g) The swap is finalized and the user can either decide to swap all tokens to the borrowToken and repay the outstanding position or simply swap a small amount to borrowToken and receive the rest of the collateral via the simple safeTransfer within _repay. Even 1 wei of borrowToken is enough to finalize the repay call. This attack can also be executed more trivially via the borrow function where a user borrows funds without having collateral at all, then reenter into liquidate and offset all just borrowed collateral to suppliers as bad-debt. The result of this action is that the whole borrowed amount was stolen. We recommend removing this refund mechanism because it is usually expected for users to provide accurate swap data such that

	the full input amount is consumed. We furthermore recommend a full audit of the swap mechanism Another concern is potential hijack of the control-flow by a malicious actor which is not the user, this actor can then liquidate the user while the position is unhealthy. This however can only happen if the swap routes through a pool with a malicious token, which is still based on user provided parameters. **The "Healthy position remains tagged" issue further inflates the feasibility of such an attack. Furthermore, priority liquidation is only catched up to disablePriorityLiquidationAbovePositionLtv which means this attack can be executed while ignoring the priority logic.**
Recommendations	We recommend either implementing a reentrancy guard for all functions which allows the AltoLeverage contract to reenter into the required functions (addCollateral/repay). This reentrancy guard will interfere with liquidation callback but this is considered as an acceptable limitation for increased security. Alternatively, a solution could be to add a reentrancy guard only to the borrow/removeCollateral and liquidate functions. Furthermore, we also recommend adding a reentrancy guard directly to the AltoLeverage contract. Please note that the potential refund of unconsumed tokens within the AltoLeverage contract exposes the same vulnerability in case the refund token is an ERC777 token. We recommend removing this refund mechanism because it is usually expected for users to provide accurate swap data such that the full input amount is consumed and the removal does not impact the UX in our opinion but further contributes to the minimalistic approach which makes the code more safe. We furthermore recommend a full audit of the swapper control-flow.

Comments / Resolution	Fixed by adding reentrancy guards on borrow/removeCollateral and liquidate. It has to be noted that tagLiquidatablePosition is still callable when a position is in an invalid state, which raises a separate issue.

Issue_04	ERC777 collateralToken OR borrowToken allows for reentrancy
Severity	High*
Description	*Important note: The Alto team confirmed that they do not ever plan to use such a token as collateralToken or borrowToken - therefore, they do not consider this as an issue. However, technically and as per source code, this issue is still considered as valid (fixed now), which is why we are required to mark it as high severity. For Alto's purposes it is however only considered as informational Whenever the borrow function is called, the state is changed and the position becomes less healthy, then the borrowToken is transferred out and the solvency check is executed at the end of the function. In case the borrowToken is an ERC777 token, one can simply borrow in the same fashion as in the above described issue and self liquidate the position, effectively draining the contract. A similar issue is present within removeCollateral in case the collateralToken is an ERC777 token: - Attacker deposits 100e18 collateral - Attacker borrows 75e18 borrow token - Attacker withdraws 100e18 collateral and reenters into the liquidate function - Due to the collateral being zero already, the collateralToSellInBorrowAssets amount becomes zero due to clamping and the full 75e18 is offset as bad debt - Attacker essentially profited with 75e18 in borrow token The "Healthy position remains tagged" issue further inflates the feasibility of such an attack. Furthermore, priority liquidation is only catched up to disablePriorityLiquidationAbovePositionLtv

	which means this attack can be executed while ignoring the priority logic.
Recommendations	Consider implementing reentrancy guards. In alignment with the leverage mechanism, these reentrancy guards should allow the AltoLeverage contract to reenter into the repay and addCollateral function only.
Comments / Resolution	Fixed by the same recommendation as the issue above, adding reentrancy guards on borrow/removeCollateral and liquidate, and removing the refund mechanism on onBorrowCallback. **Note:** But a reentrancy can still occur in onRemoveCollateralCallback via the refund mechanism if the collateral token is an ERC777. As noted on another issue it can be reentered through tagLiquidablePosition and tag a position. As a consequence if the user who tagged his own position, somehow changes the state of his own position to liquidable before the liquidation window tag expires, he can liquidate it instantly and thus bypass an important safety mechanism of the protocol. As we mentioned in the introduction, we see it as mandatory that liquidations are permissioned during the initial time to prevent abusing any potential faulty states. We therefore highly recommend not adding any ERC777 tokens as collateral token and adding a reentrancy guard to the tagLiquidatablePosition function.

Issue_05	Certain positions can never be liquidated due to truncation
Severity	**High**
Description	Whenever a borrower's position is liquidated, the very first step is to calculate the positionCollateralValue, which is basically the collateral amount converted into the borrowToken denomination: > position.collateralAssets * collateralPrice / 1e36 Note that collateralPrice is simply collateralInBorrow with 36 decimals. This calculation rounds down in an effort to ensure that the collateral valuation is against the favor of the user. This practice is important to not overstate the collateral value. However, it exposes an edge-case that can be exploited in the scenario where: position.collateralAssets * collateralPrice < 1e36 In this scenario, it will truncate down to zero, resulting in a subsequent division by zero error which fully prevents the liquidation. Example: collateralToken = DAI/FRAX (18 decimals) borrowToken = GUSD (2 decimals) price = 1e20 Arithmetic: floor((1e16-1) * 1e20 / 1e36) = 0 This can become problematic in two scenarios: a) Position with bad debt is attempted to be liquidated but due to truncation, it will not immediately enter the zero collateral

	(bad-debt branch) but requires a follow-up liquidation (which now reverts due to the above described situation). The situation where newLTV = 0 while resulting in leftover collateral stems from the way how initially remainingCollateralAfterFeeInBorrowAssets is truncated, then subsequently used to clamp collateralToSellInBorrowAssets. The rounding up of totalCollateralToTake may however not offset the previous truncation, resulting in totalCollateralToTake < positionCollateral. b) Users can deliberately create thousands of small positions which accrue interest (and thus also bad-debt at some point) and can never be liquidated
Recommendations	A simple solution would be to allow liquidations for positions with a converted collateral value of zero. However, that would mean refactoring of the liquidate function and we highly recommend against it, as it will not only potentially introduce unexpected edge-case but essentially requires a full re-audit of the liquidate function under all considered scenarios. **Therefore, our recommendation is to implement a governance-controlled function which allows governance to liquidate such positions** by simply repaying/clearing the full debt and receiving the corresponding collateral amount. The reason why this should be a governance-controlled function lies behind the reasoning that such a new liquidate mechanism may also expose unexpected edge-cases and by guarding this function, any potential attack-vector is essentially diminished.
Comments / Resolution	Fixed by introducing a governance-controlled function which simply takes the debt left of a position (that its collateral rounds down to zero so it can not be liquidated normally) and is applied as a bad debt. Then takes the collateral left and is sent to the governor. **Note**: This function will not repay any debt left, all the debt is offset

	to the suppliers. If this function is used for normal liquidable positions (allowed), it will take all the collateral and leave the debt to the suppliers, without repaying anything.

Issue_06	Liquidator can receive fee for fee due to truncation within remainingCollateralAfterFeeInBorrowAssets calculation
Severity	Medium
Description	Whenever a position gets liquidated, the fee is first deducted from the collateral amount: > uint256 remainingCollateralAfterFee = _position.collateralAssets - calc.liquidatorBonusFee and then the remainingCollateralAfterFee is converted to the corresponding value in borrow denomination: remainingCollateralAfterFeeInBorrowAssets = remainingCollateralAfterFee * oraclePrice / 1e36 (rounded down) This arithmetic operation can result in remainingCollateralAfterFeeInBorrowAssets to become zero in certain conditions. It is important to note that remainingCollateralAfterFeeInBorrowAssets is used as clamping safeguard for collateralToSellInBorrowAssets: calc.collateralToSellInBorrowAssets = Math.min(calc.collateralToSellInBorrowAssets, calc.remainingCollateralAfterFeeInBorrowAssets); The reason why this is done is to ensure the actual repay amount / seize amount does not exceed the actual existing collateral. If now, remainingCollateralAfterFeeInBorrowAssets truncates to zero, this means that essentially no repayment is provided by the liquidator. However, the liquidator will still receive totalCollateralToTake which is calculated as: totalCollateralToTake = (collateralToSellInBorrowAssets * 1e36 /

oraclePrice) + liquidatorFee

This scenario becomes possible if the collateral price decreases to a point where collateralValue does not round down to zero, but remainingCollateralAfterFeeInBorrowAssets does. In such a case, the position can be exploited repeatedly until collateralValue itself rounds down to zero.

Example:

Initial parameters:

- collateralPrice = 0.0003e36

User position:

- collateralAssets = 10,525 wei
- collateralValue = 3 wei
- borrowAssets = 2 wei

Price drops:

- collateralPrice = 0.0001e36
- collateralValue = 1 wei

Liquidation parameters:

- liquidationBaseFee = 0.05e18

Liquidation results:

- liquidationPercentage = 1e18
- newLtv = 0
- liquidatorBonusFee = 526 (ignoring protocol fee for simplicity)
- remainingCollateralAfterFee = 10,525 – 526 = 9,999 wei
- remainingCollateralAfterFeeInBorrowAssets = 0 wei (due to truncation)
- collateralToSellInBorrowAssets = 2 wei → clamped to 0 wei
- totalCollateralToTake = 526 wei (only liquidation fee)
- repayAmount = 0

Final state:

	- Collateral left = 9,999 wei - Borrow value = 2 wei In this case, the user effectively extracted part of their collateral from a bad-debt position without repaying any borrow. The attack cannot be repeated immediately, as collateralValue will now round down to zero, leading to a division-by-zero revert. However, if the user deposits just one more wei, the exploit can be repeated to extract additional collateral: **User position:** - collateralAssets = 10,000 wei - collateralValue = 1 wei - borrowAssets = 2 wei **Liquidation results:** - liquidatorBonusFee = 500 - remainingCollateralAfterFee = 9,500 wei - remainingCollateralAfterFeeInBorrowAssets = 0 wei (truncation) - collateralToSellInBorrowAssets = 2 wei → clamped to 0 wei - totalCollateralToTake = 500 wei - repayAmount = 0 **Final state:** - Collateral left = 9,500 wei - Borrow value = 2 wei The user has now extracted the maximum possible collateral without repaying any debt. To continue exploiting the system, they would need to wait for a further price decrease so that the rounding behavior repeats. It is now important to highlight that if an attacker controls both the borrower and the liquidator address and deliberately creates such a position that results in collateralToSellInBorrowAssets = 0 while the fee is paid out AND at the same time, the bad-debt branch is triggered, this may open up an attack-vector which can be

	repetitively exploited to drain a small amount of funds out of the system. Such a bad-debt branch can never be reached if remainingCollateralAfterFeeInBorrowAssets is truncated fully to zero but it can be theoretically achieved in the following course of roundings: remainingCollateralAfterFeeInBorrowAssets: truncation but non-zero result clamping of collateralToSellInBorrowAsssets with the truncated remainingCollateralAfterFeeInBorrowAssets value totalCollateralToTake: rounding up, non-even division This state basically takes advantage of the following facts: - remainingCollateralAfterFeeInBorrowAssets is truncated which means less repayment amount is required - totalCollateralToTake is rounded up so the full collateral is taken - The bad-debt branch is triggered due to full collateral take and the leftover debt (due to truncation of repayment amount) is offset to suppliers If an attacker is able to achieve this state by providing borrowing and liquidating with his two owned addresses and immediately bring his own position into a liquidatable state (which could be true in case maxLTV = maxLiquidationLTV), he might be able to repetitively exploit this mechanism and create new positions, liquidate these, receive a fee which is larger than [collateral - debt] and conquer a net-profit with each repetitive action.
Recommendations	There is no trivial fix for this issue. If remainingCollateralAfterFeeInBorrowAssets would be rounded up instead of down, the calculation of calc.collateralToSellInBorrowAssets (which is the amount required

	to meet newLTV) would now be smaller than expected due to the overstated remainingCollateralAfterFeeInBorrowAssets amount. While it might be theoretically possible to refactor the liquidate function, it is very likely that such a refactoring introduces other issues. Furthermore, a full re-audit of all control-flows and edge-cases is then required. Our recommendation is to ensure that borrowToken and collateralToken decimals do not deviate to keep the impact minimal while at the same time keeping priority liquidations with trusted addresses, such that any potential self-owned liquidation is not exploitable. Furthermore, we highly recommend a deep-dive in the above mentioned attack-vector with self-liquidation and fuzzing. The main guard against self-liquidations in combination with bad-debt remains a proper maxLTV / maxLiquidationLTV setup. But even then, there might be some hidden edge-cases which allow such an execution due to rounding directions and small amounts. **Therefore, the only real safeguard which provides a 100% guarantee against *repetitive* small liquidity exploits of any kind, remains the implementation of trusted priority liquidators which prevent immediate liquidation by malicious actors. But for this to hold, a new function must be incorporated which allows for removing a tag for a "again healthy position".**
Comments / Resolution	Acknowledged. Comments from the protocol: "*We acknowledge this, but as mentioned in the report, there's no simple fix. This can only happen if the collateral price and amount are small enough (in terms of digits) compared to oracle price precision. It can be exploited only once, and an attacker can gain at most the liquidation fee percentage of the user's collateral. For this reason, at this time, we consider it safe to leave as is.*"

Issue_07	Griefing concerns due to implementation of tagLiquidatablePosition
Severity	**Medium**
Description	In the corresponding appendix, we have explained the idea behind tagLiquidatablePosition and the liquidation allowance timeline. In the scenario where a position is tied to any address or address(0) as liquidator, only priority liquidators can liquidate a position in the first two periods, as the tagger is essentially burned.
Recommendations	Consider elaborating the design behind this implementation.
Comments / Resolution	Acknowledged, the tagger can still be burned.

Issue_08	Change of liquidation parameters can result in instantly liquidatable positions
Severity	**Medium**
Description	The maxLiquidationLTV variable can be decreased while there are already existing positions. This can result in an instant liquidation cascade.
Recommendations	Consider either not allowing to decrease maxLiquidationLTV or ensuring that such a mis-setting never happens.
Comments / Resolution	Acknowledged.

Issue_09	Invariant violation due to token donation can result in DoS of bad-debt liquidations
Severity	**Medium**
Description	The removeSupply function does not expose a check that totalSupply.assets >= totalBorrowed.assets. Usually that is not a problem because users cannot withdraw any assets which are currently borrowed out because the transfer simply reverts due to insufficient funds in the contract. However, if there is any token donation, it is actually possible to call removeSupply which then decreases totalSupply.assets without touching totalBorrowed.assets which can result in a state where totalSupply.assets < totalBorrowed.assets which is a clear invariant violation. This will then potentially result in a revert during _applyLiquidationBadDebt due to an underflow revert when subtracting totalSupply.assets.
Recommendations	Consider implementing an invariant check that totalSupply.assets >= totalBorrowed.assets within removeSupply.
Comments / Resolution	Fixed by following the recommendation

Issue_10	Healthy position remains tagged
Severity	**Medium**
Description	Whenever a position is unhealthy, it can be tagged for liquidation purposes. It is often possible that borrowers repay in time or top-up collateral which makes the said position healthy again. In such a scenario, the position indeed remains tagged and whenever in the future it becomes liquidatable again, it can be instantly liquidated by any address (if the grace periods have passed). This is a serious concern because it essentially opens up a vector to bypass the priority liquidation mechanism. We have elaborated the requirement for the priority liquidation to support the protocol security. **Please note that due to this fact, the reentrancy attack with self liquidation is even possible in case of priorityLiquidation being enabled. Furthermore, this issue can be combined with potential self-liquidate vulnerabilities to bypass the grace period by artificially making a position unhealthy just to tag it, then repay debt and remain tagged, just to in the future trigger an immediate self-liquidation**
Recommendations	Consider implementing a function which allows untagging any previously tagged positions which are now healthy again.
Comments / Resolution	Partially fixed. A healthy position will remain tagged, but once the liquidationWindowTag expires, it must be tagged again, as neither the tagger nor normal liquidators (only priority liquidators) can liquidate it thereafter. However, if the position becomes unhealthy again before the liquidation window expires, then the tagger or any normal liquidator can liquidate it immediately, effectively bypassing the intended priority-liquidator mechanism. A possible attack can still be executed through reentrancy:

	1. Attacker borrows with leverage and reenters to tag his own position, as at that point it will be liquidatable (borrow > collateral). 2. Once the transaction finishes, the position becomes healthy again, meaning it cannot be liquidated normally. 3. If the attacker somehow brings their position to a liquidatable state within the liquidationWindowTag and after the priorityLiquidationGracePeriod, they can liquidate their own position instantly, bypassing the priority-liquidator mechanism.

Issue_11	tagLiquidatablePosition reentrancy allows for faulty tagging
Severity	**Medium**
Description	As mentioned in the issue resolution above, tagLiquidatablePosition can be reentered through the refund mechanism of onRemoveCollateralCallback when the collateral token is an ERC777. However, it can also be reentered via the attack path described in the issue 'Reentrancy path via swapper can be exploited to offset bad debt to suppliers'. This enables a user to tag their own position even if it is healthy by the end of the transaction. If the user manages to push their position into a liquidatable state before the tag window expires, they can bypass the priority liquidator mechanism.
Recommendations	Consider adding a nonReentrant modifier inside tagLiquidadablePosition.
Comments / Resolution	

Issue_12	Outsourcing of decimal normalization to Oracle raises concerns
Severity	**Low**
Description	Currently, there is no way to validate the correctness in the scenario where collateralToken and borrowToken decimals deviate (e.g. 6 and 18). Essentially, if the price would be 1, it would then require to return an oracle value of 1e24 and 1e48 to normalize the decimal mismatch.
Recommendations	Consider carefully evaluating that scenario in the external Oracle contract.
Comments / Resolution	Acknowledged

Issue_13	Enforcement of fee will increase bad-debt in positions with LTV >= 1e18
Severity	**Low**

Description	The liquidation fee is enforced even if a position is already above LTVForCompleteLiquidation. This means that this fee application will inherently result in bad debt in case of that specific LTV:
	
	Example:
	
	- Near LTV = 1e18 and base fee application, no bonus fee
	  - Collateral = 100e18
	  - Debt = 99e18
	  - currentLTV = 0.99e18
	  - liquidationPercentage = 1e18
	  - bonusLTV(1e18) = 0
	  - Now calculate the collateral after fee
	  - 100e18 - ((100e18 * 0.03e18) / 1e18)
	  - 97e18
	
	The result of this is that only 97e18 fee is repaid and bad debt becomes 2e18

Recommendations	Consider acknowledging this issue, fixing it would mean refactoring the liquidation mechanism. To reduce the likelihood of this happening, we furthermore recommend to keep the liquidationMaxLTV/LTVForCompleteLiquidation reasonably small compared to the maximum bonus and base fee, which means even in late liquidations, the fee will not trigger bad-debt
Comments / Resolution	Acknowledged

Issue_14	Interest can be withdrawn before actually repaid
Severity	**Low**
Description	Most lending protocols apply interest before it has been repaid which is a valid choice to do. However, it must be kept in mind that such a mechanism increases the likelihood of insolvency, as essentially funds can be withdrawn which have not been provided yet. Furthermore, if there is any mechanism to prevent liquidations while being in bad-debt, a user can deliberately accumulate interest with a supply position while not paying it with the liquidatable position.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged

Issue_15	Frontrun of bad-debt application allows for dodging fee
Severity	**Low**
Description	Whenever a liquidation is happening which offsets bad-debt, suppliers will bear the loss. It is trivially possible to inspect the mempool for such executions and simply withdraw before the application.
Recommendations	A potential solution would be to implement a withdrawalDelay after addSupply has been called. However, that would expose a griefing vector due to the onBehalf mechanism. We recommend acknowledging this issue as a design choice.
Comments / Resolution	Acknowledged

Issue_16	Repay/Liquidation concerns during paused market
Severity	**Low**
Description	Whenever a market is paused, it is not possible to repay debt or liquidate a position. This can result in bad debt and also in immediate liquidations post-unpause.
Recommendations	Consider only using the freeze feature, as this would still allow to repay debt and liquidate positions.
Comments / Resolution	Acknowledged

Issue_17	Exchange rate concerns in case of ERC4626 token as borrowToken
Severity	**Low**
Description	In case an ERC4626 token is used as borrowToken, it can happen that the whole supply is provided via addSupply, and an attacker then borrows all of it. If an attacker owns the full supply from an ERC4626 vault, he can redeem it fully to get out the corresponding underlying token just to deposit it again to reset the exchange rate. The new deposit will then require less underlying token to match the corresponding erc4626 share amount which has been borrowed, allowing the attacker to essentially steal the erc4626 vault's full profit in one transaction, while still honoring the debt payment.
Recommendations	Consider not adding ERC4626 tokens as borrowToken.
Comments / Resolution	Acknowledged

Issue_18	Missing emergency change possibility during setIRM in case of accrual revert
Severity	**Low**
Description	The setIRM function forces accrual using the current IRM. In the scenario where the IRM is broken due to an unforeseen vulnerability. the setIRM function may revert due to the revert of the _accrueInterest function, effectively requiring a protocol upgrade.
Recommendations	Consider allowing to call the setIRM function with an ignore boolean parameter that allows for changing the IRM without calling _accrueInterest on the existing IRM.
Comments / Resolution	Fixed by following the recommendation

Issue_19	Potential overshoot of newLTV due to truncation
Severity	**Informational**
Description	The newLTV value can result in a very low value. For example it can become 7 in case of a liquidationPercentage of 999999999999999990. In the scenario where it is indeed such a low value, the collateralToSellInBorrowAssets will truncate the first multiplication and result in a higher value as expected (even above the positionBorrowedDebt due to the division of 1e18-7): - (debt - (collateral * newLTV / 1e18)) * 1e18 / (1e18 - newLTV) which then means, while indeed the newLTV should be 7 and not zero, it will now become zero due to the truncation and is fully liquidated. In the real-world this is not an issue as it is against the favor of the user and safeguards such as debt clamping are anyways present.
Recommendations	Consider acknowledging this impact.
Comments / Resolution	Acknowledged

Issue_20	Confusing divideWithRounding implementation
Severity	**Informational**
Description	The divideWithRounding implementation is implemented as follows: *uint256(calc.liquidatorBonusFee).divideWithRounding(liquidationConfiguration.protocolFeePercentage, MATH_PRECISION, true);* It basically indicates its doing a division. However, the math is as follows: - liquidatorBonusFee * protocolFeePercentage / 1e36 Even though this is mathematically correct, it indicates that a division with protocolFeePercentage is done (which would be false).
Recommendations	We do not recommend a change but rather adding natspec which clearly describes the logical behavior.
Comments / Resolution	Fixed by following the recommendation. The natspec has been added to FixedPointMath.sol.

Issue_21	Opening fee is not incorporated into totalSupply.assets > totalBorrowed.assets invariant check
Severity	**Informational**
Description	During borrowing, a fee is taken, this fee is expected to be repaid by the borrower. The way how this fee is applied is a simple reflection of the addSupply function. There is however an inconsistency in the safeguard: *if (totalBorrowed.assets > totalSupply.assets)* This is actually executed before the fee is reflected, the fee reflection would basically just be an increase of totalSupply.assets (and shares) Basically, one should theoretically be able to borrow up to totalSupply.assets (including the fee). But this safeguard accidentally implements a small reserve (until feeRecipient has withdrawn), which is net-positive from a security perspective. Furthermore, given the current missing invariant check (totalSupply.assets >= totalBorrow.assets) within the removeSupply function, the current implementation serves as a safeguard. If the invariant check would have been post-fee application, the feeRecipient could actually withdraw the fee (because it remains sitting in the contract) while the actual borrower has not repaid it fully, resulting in a state where totalSupply.assets < totalBorrow.assets.
Recommendations	We do not recommend a change because that strict check is positive for the overall code safety, it is simply an information for the developer
Comments / Resolution	Acknowledged

Issue_22	Lack of reserve implementation
Severity	**Informational**
Description	Currently, there is no reserve which prevents borrowing up to the totalSupply.asset value. While this is not a direct concern, it is worth pointing out that suppliers may not be allowed to withdraw supplied borrowTokens.
Recommendations	Consider accepting this risk and communicating it with users.
Comments / Resolution	Acknowledged

Issue_23	Inheritance of Pausable contract without gap
Severity	**Low**
Description	The Pausable contract does not expose a gap variable which means in case of upgrades where new variables are added to the Pausable contract, this can shift the storage layout and result in corruption.
Recommendations	Consider adding a _gap to the Pausable contract.
Comments / Resolution	Acknowledged

Issue_24	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 these tokens.
Comments / Resolution	Acknowledged

AltoBorrowMarket

The AltoBorrowMarket is an extension of the AltoBaseMarket and exposes the borrowOpeningFee which is settable by the admin and is simply considered whenever a borrow is executed, as a part of the borrowed assets is then assigned to the feeRecipient and deducted from the transfer to the borrower.

Appendix: borrowOpeningFee

The borrowOpeningFee is taken on each borrow execution and is a specific (governance-configurationable) percentage on the overall borrowed amount.

The fee is then simply decreasing the transferred out borrow amount and allocated to the feeRecipient by mirroring the logic within addSupply.

Privileged Functions

- setBorrowOpeningFee

Issue_25	Race condition for setBorrowOpeningFee can result in unexpected loss for borrowers
Severity	**Medium**
Description	Governance can change the openingFee to any parameter up to 1e18. This can raise serious race-condition concerns in case some users have just submitted a borrow transaction which has not been executed yet. If the fee is now increased, users will experience the new borrowing fee while expecting the older, lower value, resulting in a loss of funds.
Recommendations	Consider setting a reasonable limitation for the borrowingFee as well as communicating any fee increase upfront with the community. Alternatively, it is also possible to add an extra minBorrowAmount parameter which is then sanity checked against assetsToSend and revert if undercut.
Comments / Resolution	Acknowledged

AltoLeverage

The AltoLeverage contract is an externally deployed contract and each AltoBaseMarket contract has a corresponding AltoLeverage contract (if leverage is enabled).

This contract simply allows for leverage and deleverage actions in the known manner.

Appendix: Leverage Mechanism

In lending protocols, users can leverage their position by looping, which means a repetitive sequence of depositing, borrowing, swapping. This would look as follows:

Start (before looping):

- Collateral = 100, Debt = 0, LTV = 0.000

Loop 1

- Borrow to 75% of 100 ⇒ +75 debt, swap to collateral ⇒ +75 collateral
- State: Collateral 175, Debt 75, LTV 75 / 175 = 0.428571

Loop 2

- Max debt at 175 is 0.75×175 = 131.25 ⇒ additional borrow 56.25, swap ⇒ +56.25 collateral
- State: Collateral 231.25, Debt 131.25, LTV 131.25 / 231.25 ≈ 0.567568

Loop 3

- Max debt at 231.25 is 173.4375 ⇒ additional borrow 42.1875, swap ⇒ +42.1875 collateral
- State: Collateral 273.4375, Debt 173.4375, LTV ≈ 0.634204

Loop 4

- Max debt at 273.4375 is 205.078125 ⇒ additional borrow 31.640625, swap ⇒ +31.640625 collateral
- State: Collateral 305.078125, Debt 205.078125, LTV ≈ 0.672268

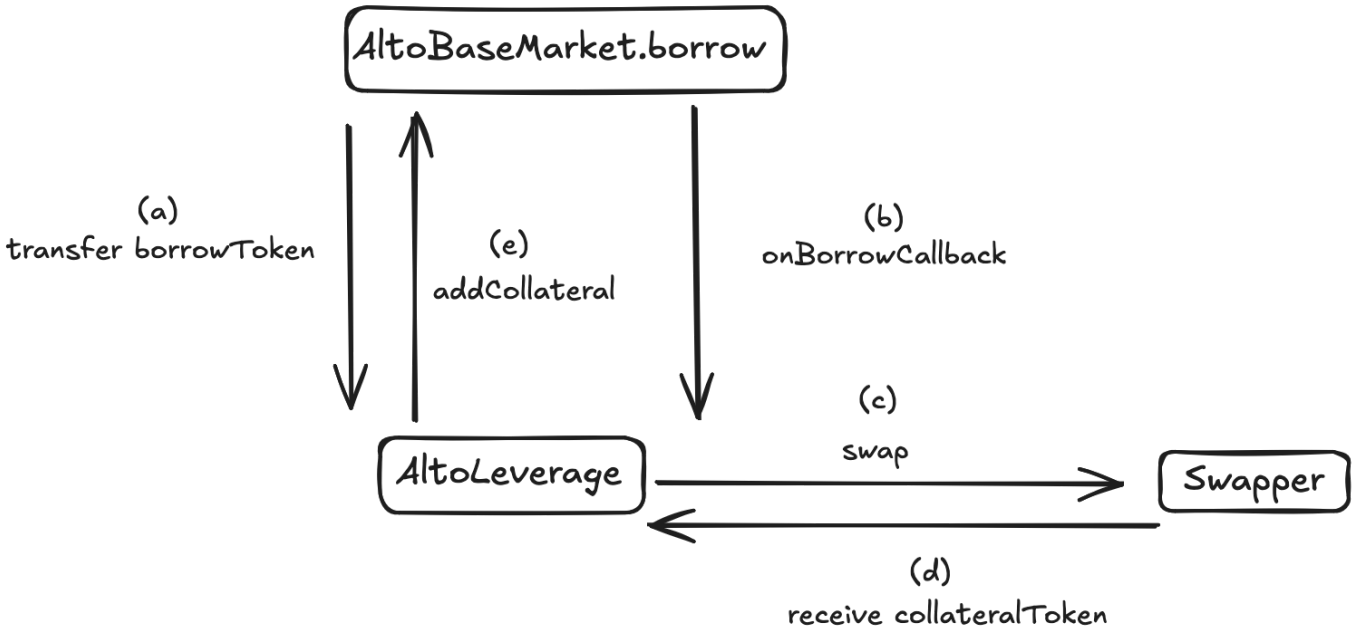
Loop 5

- Max debt at 305.078125 is 228.80859375 ⇒ additional borrow 23.73046875, swap ⇒ +23.73046875 collateral
- State: Collateral 328.80859375, Debt 228.80859375, LTV ≈ 0.696029

In practice, this will look slightly different due to slippage and fees.

Using this technique, it is possible to receive a higher collateral amount than the nominal owned tokens.

The leverage mechanism is a shortcut to bypass this looping, as it simply allows to borrow a specific specific amount of funds, swap it to the collateral asset and deposit it as collateral. It is required to have already a specific collateral amount, as otherwise only with borrowing/swapping/depositing, the LTV would be >= 1e18 (unless some arbitrage was executed during the swap which results in a higher value of collateral tokens than borrowed, after the swap). The whole control-flow can be illustrated as follows:



Appendix: Deleverage Mechanism

Similar to the leverage mechanism, a shortcut allows for deleveraging a position without unlooping it. Traditional unlooping for the above mentioned position would be as follows:

Start: C0 = 328.80859375, D0 = 228.80859375, LTV ≈ 0.696029

Unwind 1

- w1 = C0 − D0/0.75 = 328.80859375 − 305.078125 = 23.73046875
- Withdraw 23.73046875 ⇒ interim (305.078125, 228.80859375) with LTV = 0.75
- Repay 23.73046875 ⇒ State: C1 = 305.078125, D1 = 205.078125, LTV ≈ 0.672268

Unwind 2

- w2 = 305.078125 − 205.078125/0.75 = 31.640625
- Interim (273.4375, 205.078125) @ 0.75 → Repay
- State: C2 = 273.4375, D2 = 173.4375, LTV ≈ 0.634204

Unwind 3

- w3 = 273.4375 − 173.4375/0.75 = 42.1875
- Interim (231.25, 173.4375) @ 0.75 → Repay
- State: C3 = 231.25, D3 = 131.25, LTV ≈ 0.567568

Unwind 4

- w4 = 231.25 − 131.25/0.75 = 56.25
- Interim (175, 131.25) @ 0.75 → Repay
- State: C4 = 175, D4 = 75, LTV ≈ 0.428571

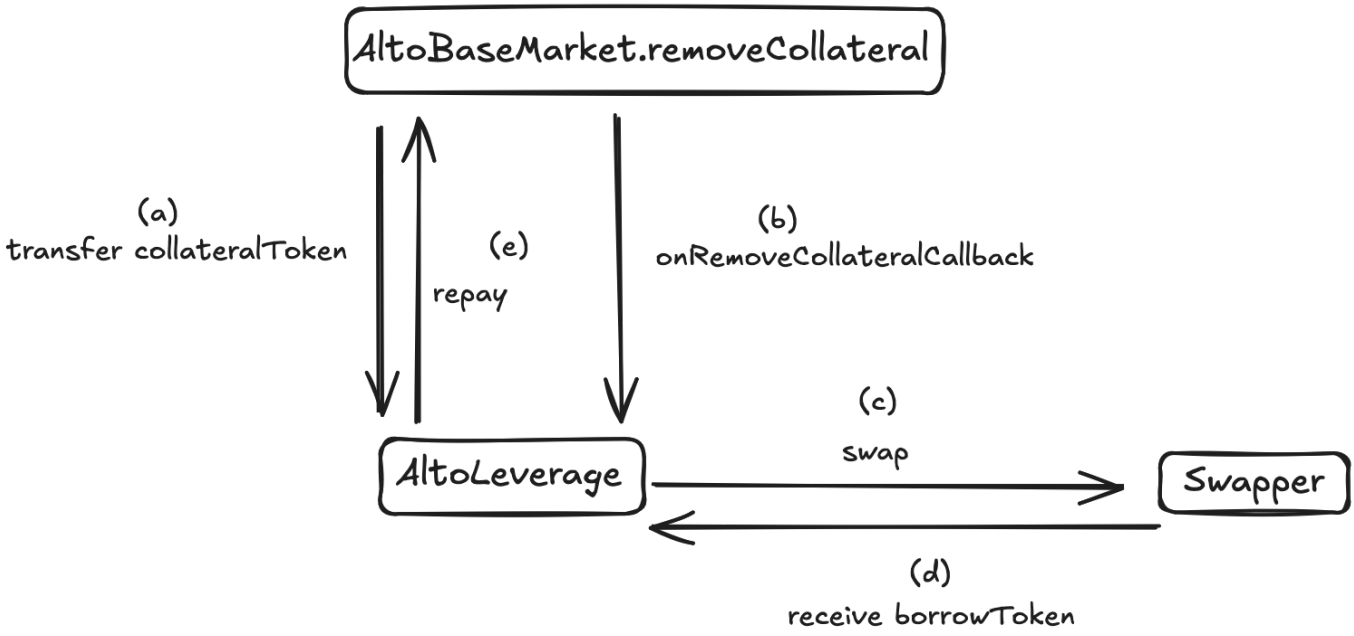
Unwind 5

- w5 = 175 − 75/0.75 = 75
- Interim (100, 75) @ 0.75 → Repay
- State: C5 = 100, D5 = 0, LTV = 0

The deleverage mechanism implements a shortcut where the collateral can be removed, sold and the debt repaid in the same transaction. With the above example:

- collateral = 328
- debt = 228

The removeCollateral function is called with 228 collateral, which is swapped to the borrowToken and repaid. The end-result is a collateral amount of 100 and no debt.



Privileged Functions

- setSwapper
- setMarket
- onBorrowCallback
- onRemoveCollateralCallback

Core Invariants:

INV 1: The exact assets amount must be received before onBorrowCallback and onRemoveCollateralCallback is invoked

INV 2: During _repay and full repayment, userBorrowedAssets must be equal to assets return value of Market.repay

Issue_26	Potential multi-usage concern
Severity	**Low**
Description	The AltoLeverage contract is meant to be used with multiple authorized markets, while we could not find a way to exploit this mechanism, there are concerns about cross-market reentrancy attacks where one leverage execution could enter into another market which triggers another leverage execution while the first leverage execution is still running.
Recommendations	We recommend deploying one AltoLeverage contract for each market to fully mitigate this risk.
Comments / Resolution	Acknowledged by following the recommendation.

Issue_27	Repayment may revert in case amountOut = userBorrowedAssets if exchange rate becomes negative during unexpected edge-cases
Severity	**Informational**
Description	In the scenario where amountOut is equal to userBorrowedAssets, while userBorrowedAssets has been calculated as follows: > userBorrowedAssets = borrowShares * totalBorrowed.assets / totalBorrowed.shares (up) userBorrowedAssets = ceil(75 * 100 / 200) = 38 , it will call the repay function with amountOut as parameter which then calculates the shares amount as follows: > shares = assets * totalBorrowed.shares / totalBorrowed.assets (down) floor(38 * 200 / 100) = 76 Please note that this example is done with exchangeRate < 1 and no virtual shares. Thus, in some cases the rounding down is insufficient to offset the previous ceiling and thus it is possible to overestimate the shares amount which will result in a revert.
Recommendations	Consider adding the equal case to the first condition, as it will then exactly pull amountOut
Comments / Resolution	Fixed by following the recommendation

Issue_28	Note: onBorrowCallback/onRemoveCollateralCallback
Severity	Informational
Description	The onBorrowCallback function has been adjusted in an effort to remove a potential token refund in case where not all input tokens have been consumed by the swap. This reduces potential reentrancy concerns. It has to be mentioned that it now must be ensured that the swap control-flow(which is out of scope) should consume all input tokens. Furthermore, a small note must be made that the refund mechanism for the onRemoveCollateralCallback is still existent, which forms an inconsistency compared to the onBorrowCallback function and still exposes reentrancy concerns. We understand that in case of unleveraging, there is anyways a transfer-out possibility in case the received output amount is larger than what has been borrowed. Therefore, it is a reasonable business decision to give users the right to not swap all their collateral (and swap only what's indeed required). Though, the actual difference transfer in case of full repayment happens after the position has been fully repaid already, while the refund of the unconsumed input token happens during an invalid state. However, it should be kept in mind that an ERC777 token as collateral now still exposes certain risks.
Recommendations	Consider keeping this in mind
Comments / Resolution	

AdaptiveCurveIRM

The AdaptiveCurveIRM contract is an advanced plug-in IRM which can be used by the Alto Lending protocol. It uses a customized interest rate model that is forked from Morpho's AdaptiveCurveIRM contract:

https://github.com/morpho-org/morpho-blue-irm/blob/main/src/adaptive-curve-irm/AdaptiveCurveIrm.sol

The rate calculation mechanism is 1:1 identical and the changes to the Morpho codebase are limited to the following factors:

- Incorporation of pausing implementation
- Ability to customize parameters
- Interest calculation within the contract and not within the market
- Each market has its own AdaptiveCurvIRM contract

Appendix: Rate Calculation

The Adaptive Curve IRM is designed to automatically adjust the borrow rate of a lending market based on:

- how utilized the market is (ratio of borrowed to supplied liquidity)
- how long the utilization stays above or below a target level
- and how aggressively the protocol wants to adapt

This mechanism ensures that markets remain balanced and discourages excessive borrowing when liquidity is scarce (high utilization) and encourages borrowing when liquidity is abundant (low utilization). Essentially, it attempts to shift the utilization to the target.

Privileged Functions

- onMarketPause
- setIRMConfig
- updateInterestRate

Core Invariants:

INV 1: If the market is paused, interest must be accrued up to block.timestamp

INV 2: If the market is unpaused, lastUpdate must be set to block.timestamp

INV 3: If the market is unpaused and the borrowRate is updated, interest must be accrued up to block.timestamp

Issue_29	Rate moves to lowest bound in case of no utilization
Severity	Informational
Description	When a market remains unutilized for an extended period (totalSupply > 0 and totalBorrowed = 0), the rateAtTarget will gradually decrease until it reaches minRateAtTarget. Depending on the irmConfiguration, if the market then suddenly becomes overutilized, the rate will take days or even weeks to adjust back to normal levels. During this adjustment phase, borrowers may pay significantly less interest than expected, despite the market being overutilized. While this behavior is expected from IRM, if adjustmentSpeed and curveStepness are not properly configured (e.g., both set too low), borrowers may exploit this dynamic to minimize interest payments and maximize their gains. Scenario: - rateAtTarget: 0.1e18 - totalSupply: 1000e18 - totalBorrowed: 0 - utilization: 0 - targetUtilization: 0.7e18 Over time, due to zero borrowing activity, the rate decreases and hits the minimum: - rateAtTarget: 0.001e18 A borrower then enters, and the market becomes overutilized: - totalSupply: 1000e18 - totalBorrowed: 900e18 - utilization: 0.9e18 At this point, the borrower begins paying interest, but at a much lower rate than expected, since the IRM needs time to adjust upward. While a low curveStepness will increase the interest rate due to overutilization, it will still be far below the level it would have been if the market had already adjusted. Once the interest rate eventually increases to normal levels, the borrower can repay, wait for the rate to decrease again, and repeat the cycle.

	This scenario will be plausible when adjustmentSpeed and curveStepness are set to low values.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged

Issue_30	Inconsistent rate when utilization meets targetUtilization
Severity	**Informational**
Description	In a market where utilization == targetUtilization, the interest rate remains constant and does not adjust further. The issue is that once the target utilization is reached, the rate becomes fixed at its current value, regardless of whether it is high or low. For example, if the target utilization is reached at a rate of 2e18, it will remain at that value. The same will happen if the target is reached at 0.001e18. This behavior may be perceived as unfair for both parties (depending on the scenario), as the resulting rate depends on when the target utilization was reached rather than on a consistent equilibrium mechanism.
Recommendations	Consider acknowledging this issue
Comments / Resolution	Acknowledged

Issue_31	Unsupplied market will lower the rate
Severity	**Informational**
Description	In a market where totalSupply = 0, the interest rate gradually decreases over time. This behavior, though impractical, may disincentivize suppliers from entering the market. So when the market is supplied, it will use a lower rate than the initially configured one due to the inactivity. Conceptually, it can also be said that when there is no supply, it can make sense to increase the rate to attract new suppliers rather than decrease it.
Recommendations	This can be considered a design decision.
Comments / Resolution	Acknowledged

Issue_32	Permissionless setMarket function can result in griefing during non-atomic deployment and initialization
Severity	**Informational**
Description	Currently, after deployment it can happen that an attacker hijacks the IRM via the setMarket function, which makes it non-functional for the market.
Recommendations	This is considered as a non-issue because in the worst scenario, a new AdaptiveCurveIRM contract can be deployed.
Comments / Resolution	Acknowledged

FixedRateIRM

The FixedRateIRM contract is a simple plug-in IRM which can be used by the Alto Lending protocol. It uses a fixed rate interest model based on the current borrowRate and the outstanding borrowed amount, while correctly compounding any interest based on the time passed since the last update.

Privileged Functions

- setBorrowRate
- onMarketPause
- updateInterestRate

Core Invariants:

INV 1: If the market is paused, interest must be accrued up to block.timestamp

INV 2: If the market is unpaused, lastUpdate must be set to block.timestamp

INV 3: If the market is unpaused and the borrowRate is updated, interest must be accrued up to block.timestamp

Issue_33	Permissionless setMarket function can result in griefing during non-atomic deployment and initialization
Severity	**Informational**
Description	Currently, after deployment it can happen that an attacker hijacks the IRM via the setMarket function, which makes it non-functional for the market.
Recommendations	This is considered as a non-issue because in the worst scenario, a new FixedRateIRM contract can be deployed.
Comments / Resolution	Acknowledged

AssetShareConversionMath

The AssetShareConversionMath contract is a simple helper contract which is used for converting between assets and shares in an ERC4626-like fashion, enabling different rounding directions.

Privileged Functions

- none

Core Invariants:

INV 1: All helper functions must be used to round against the favor of the user

No issues found.