BAIL security

Hyperdrive Markets

FINAL REPORT

March ‘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	Audit - Hyperdrive Markets
Website	hyperdrive.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/ambitfi/hyperdrive-contracts/tree/589278ce69c3c4f15f7717338926abd39083afc9/packages/lending/contracts/protocol/markets
Resolution 1	https://github.com/ambitfi/hyperdrive-contracts/tree/67b37cc9d127c1772c3069eecee16e81adaa17e8/packages/lending/contracts/protocol/markets

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	7	5	1	1	
Medium	15	9		5	1
Low	15	6		9	
Informational	9	5		4	
Governance	1			1	
Total	47	25	1	20	1

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

Bailsecurity does not consider this deployment ready !

Market

CollateralLib

The CollateralLib contract is a library contract which is imported by the Market contract and facilitates all collateral related actions, such as:

a) Registry and configuration of collateral assets

b) ETH supply logic

c) ERC20 supply logic

d) ETH withdraw logic

e) ERC20 withdraw logic

Appendix: Supply Flow

Users can supply collateral via the supplyCollateral function for standard ERC20 tokens and via the supplyCollateralETH function for native ETH. The supplyCollateralETH function simply incorporates a wrapper before the actual supply call which wraps ETH to WETH and then executes the supply call.

The flow for this function is as follows:

a) The beforeSupplyHook is invoked
b) collateralSupplied[token] is increased by the supply amount and the maxSupply check is executed
c) The token is added to the user's userCollateralTokens[account] set
d) The userCollateralSupplied[account][token] mapping is increased by the supplied amount

e) The token is transferred in (if it was a ERC20 supply). For the native scenario, WETH is already sitting in the contract
f) The afterSupplyHook is triggered

Appendix: Withdraw Flow

Similar to the supply flow, users can also withdraw assets which they have provided as collateral, as long as their position remains healthy. This can be done via the withdrawCollateral function to withdraw ERC20 tokens or via the withdrawCollateralETH function to withdraw native ETH from the WETH position.

The flow for the ERC20 function is as follows:

a) Caller sanity check
b) Fetching asset configuration
c) Ensure user cannot withdraw more than supplied
d) Invoke beforeWithdrawHook
e) Decrease collateralSupplied[token] by the withdrawn amount
f) Decrease userCollateralSupplied[account][token] by the withdrawn amount
g) Optional (if user has no supply left): Remove token from userCollateralTokens[account] set
h) Transfer amount out to recipient
i) Invoke afterWithdrawHook
j) Account health check

The flow for the ETH function is as follows:

a) Fetching asset configuration
b) Ensure user cannot withdraw more than supplied
c) Invoke beforeWithdrawHook
d) Decrease collateralSupplied[token] by the withdrawn amount
e) Decrease userCollateralSupplied[account][token] by the withdrawn amount
f) Optional (if user has no supply left): Remove token from userCollateralTokens[account] set
g) Transfer amount out to msg.sender
h) Invoke afterWithdrawHook
i) Unwrap WETH to ETH
j) Transfer native ETH out to msg.sender

Appendix: Configuration of collateral assets

The contract allows the owner to add several ERC20 tokens as collateral token which can then be used to borrow the main asset. Specifically, this can be done via the addOrUpdateAsset function which is only callable by governance and is using the CollateralAsset struct as parameter which contains the following information.

a) token: collateral token address
b) maxLTV: LTV in BPS
c) priceOracle: oracle address
d) liquidationDiscount: discount for liquidation in BPS
e) maxSupply: max allowed collateral amount
f) hooks[]: array of structs with Hooks
    i) callback: address of the hook
    ii) options: flags to determine hook execution

The collateral info is fetched via:

a) findAsset
b) tryFindAsset
c) hydrateAsset

In an effort to optimize the gas consumption, not all values are returned but only those that the corresponding flags matched. If a flag does not match, the default type for this variable is returned. For example an address returns address(0).

a) During supplyCollateral/ETH, only FIELDS_HOOKS and FIELDS_MAX_SUPPLY is consulted
b) During withdrawCollateral/ETH, only FIELDS_HOOKS is consulted
c) During isHealthy, only FIELDS_PRICE_ORACLE and FIELDS_MAX_LTV is consulted

Core Invariants:

INV 1: toLiabilities must always be called with updated state

INV 2: findAsset must revert if there is no existing token

INV 3: User can never withdraw collateral more than what was provided

INV 4: maxSupply must never be exceeded

INV 5: collateralSupplied must match all individual userCollateralSupplied amounts

Privileged Functions

- none

Issue_01	Incorrect double spending via withdrawCollateralETH can be abused to drain the contract
Severity	**High**
Description	The contract has two dedicated flows for withdrawing collateral, the withdrawCollateral and withdrawCollateralETH function. The latter function incorrectly double spends the WETH/ETH transfer as it will call the internal withdrawCollateral function which transfers out WETH to the recipient while at the same time the native ETH is transferred out at the end of the function, allowing a user to drain all WETH in the contract.
Recommendations	Consider setting the recipient address to address(0) for the internal withdrawCollateral call. The event emission must be adjusted as well.
Comments / Resolution	Resolution 1: (Resolved), the withdrawCollateralETH function has been refactored, as well as: - supplyCollateral - supplyCollateralETH - withdrawCollateral The double spending bug has been fixed in the new version. However, all flows must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_02	Incorrect event emission due to native ETH logic
Severity	**Informational**
Description	The supplyCollateral function emits the following event: *emit IMarket.SupplyCollateral(account, token, $.marketId, amount, supplier, msg.sender, block.timestamp);* This will be incorrect in the native scenario as supplied will be address(0).
Recommendations	Consider adjusting the event parameters.
Comments / Resolution	Resolved.

Issue_03	Violation of checks-effects-interactions during withdrawCollateralETH
Severity	**Low**
Description	Within withdrawCollateralETH, the native ETH is transferred out before the isHealthy check, while this seems to be theoretically correct because the isHealthy check should always be executed after all operations, it still violates the checks-effects-interactions pattern. Since we simply recommend adding reentrancy guards to all external functions, we did not dig deeper into any sophisticated exploit flows.
Recommendations	Consider adding a reentrancy guard to all external functions (In the Market contract)
Comments / Resolution	Resolved.

Issue_04	Setting of found.token in case of existing token is redundant
Severity	**Informational**

Description	The addOrUpdateAsset function allows for adding a new asset or updating an asset's configuration in case it is already existing. In the latter case, it sets found.token to asset.token, which is basically the same:
	
	require(asset.token == found.token);
	found.token = asset.token;
	
	This practice is redundant.

Recommendations	Consider removing the redundant double setting.
Comments / Resolution	Resolved.

LiquidationLib

The LiquidationLib contract is a library contract which is imported by the Market contract and facilitates liquidation related actions such as:

a) Liquidation via seizedAmount
b) Liquidation via repayAmount
c) Absorption of bad debt

Appendix: Liquidation via seizedAmount

The first liquidation method allows addresses with the LIQUIDATOR_ROLE to liquidate unhealthy accounts by providing a specific repayAmount and array of tokens. The logic will then loop over all tokens and grant the corresponding seizedAmount for the desired repayAmount. For example, if a user provides more than 1 token in the array and the supplied amount for the first token is insufficient to cover the repayAmount, the next token in the array will be considered to grant the leftover amount to the liquidator. This can be done with multiple tokens. This function is convenient if a specific desired amount should be seized. It will however always revert if the necessary repayAmount is larger than the account's debt, in that scenario, the liquidator must seize less.

The flow for this function is as follows:

a) Accrue interest
b) Ensure account is liquidatable
c) Clamp repayAmount in case it is higher than accounts total debt
d) Execute loop over tokens
    i) Calculate seizedAmount for repayAmount
        1) seizedAmount = repayAmount * 1e27 / discountedPrice
        2) In case seizedAmount is larger than what the user has supplied
            (a) Calculate repayAmount based on maximum seizedAmount
                (i) repayAmount = seizedAmount * discountedPrice / 1e27
        3) Return repaidAmount and seizedAmount
    ii) If not all repayAmount has been consumed AND there are more tokens in the array, continue with the same logic in the loop to consume the rest of the repayAmount
e) Determine how much has been repaid and cast all seizedAmounts to the corresponding repayAmount

f) Fetch account's liabilities
g) Invoke beforeLiquidateHook
h) Transfer repayAmount from liquidator in
i) Calculate shares to deduct from account
    i) shares = repaidAmount * totalSupply / totalBorrowAssets
j) Decrease totalBorrowAssets by repaidAmount
k) Decrease totalBorrowShares by shares
l) Decrease userBorrowShares[account] by shares
m) Loop over all tokens
    i) Decrease userCollateralSupplied[account][token[i]] by seizedAmount
    ii) Transfer out seizedAmount if non-zero
n) Invoke afterLiquidateHook

Appendix: Liquidation via repayAmount

The second liquidation method allows addresses with the LIQUIDATOR_ROLE to liquidate unhealthy accounts by providing an array of seizedAmounts and array of tokens. The logic will then loop over all tokens to calculate the necessary repayAmount from the seizedAmount and the discountedPrice:

> seizedAmount * discountedPrice / 1e27

The flow for this function is as follows:

a) Accrue interest
b) Ensure account is liquidatable
c) Loop over all tokens in the array
    i) Clamp seizedAmounts in case it is larger than what the user has supplied
    ii) Calculate repaidAmount from seizedAmount and discountedPrice
        1) > seizedAmount * discountedPrice / 1e27
d) Continue with the next token in the loop until all tokens and their corresponding seizedAmount have been consulted and the repaidAmount has been fully aggregated
e) Fetch account's liabilities
f) Ensure repaidAmount is not above liabilities (this will never revert: INVARIANT)
g) Invoke beforeLiquidateHook
h) Transfer asset from liquidator in
i) Calculate shares to decrease from borrower
    i) shares = repaidAmount * totalSupply / totalBorrowAssets
j) Decrease totalBorrowAssets by provided repayAmount

k) Decrease totalBorrowShares by decreased share amount
l) Decrease totaalUserShares[account] by decreased share amount
m) Loop over all tokens
    i) Decrease userCollateralSupplied[user][token[i]] by seizedAmount
    ii) Transfer seizedAmount to liquidator
n) Invoke afterLiquidateHook

Appendix: Bad-debt handling

There are several scenarios where an account can accrue bad debt. To illustrate this, we will provide two examples:

Example A: Standard bad-debt situation where the USD debt grows higher than the USD collateral. Even without liquidation discount, it now becomes impossible to fully liquidate the position

Example B: Situation where the USD debt of a user is above the LTV and the position is considered as liquidatable but the liquidation discount would exceed the collateral in case the position is fully liquidated.

Collateral = 100e18
Debt = 80e18
LTV = 0.75
Discount = 22%

The liquidator attempts to repay the full debt of 80e18 which results in a seizedAmount of 102.5e18. This will not work since the collateral amount is only 100e18. Thus the liquidator will only receive 100e18 and repay the corresponding amount of 78e18. Bad debt of 2e18 is the result

Among these two examples, there are multiple other potential scenarios. In an effort to counter these scenarios, an absorb function has been implemented which simply forgives all debt from a user while decreasing the totalReserveAsset amount and even the totalSupplyAsset amount. In the scenario where the debt is even larger than both amounts, it will be considered as toxic amount, which is theoretically a part of the amount which has been paid as interest to all depositors (which was withdrawn) before the absorb call. The underlying reason for this lies in the fact that any interest which is accrued immediately increases the deposit/withdraw exchange rate - even if no actual interest has been repaid yet.

The flow for this function is as follows:

a) Accrue interest
b) Ensure the account is liquidatable
c) Fetch liabilities from the account
d) Calculate how much can be taken from totalReserveAssets
e) Decrease totalReserveAssets based on the amount which can be absorbed. In the ideal case, the totalReserveAssets amount should be high enough to absorb the debt
f) In the scenario where there is leftover debt which was not absorbed, attempt to decrease totalSupplyAssets by the corresponding amount
g) In the scenario where also the totalSupplyAssets amount is not sufficient, the leftover will be considered as toxic amount which is simply emitted via an event. At this point, no asset is left in the contract but potentially ERC20 vault shares with zero value are left.

Core Invariants:

INV 1: It must never be possible to seize more than an account has supplied

INV 2: Only unhealthy accounts can be liquidated

INV 3: Only addresses with the LIQUIDATOR_ROLE can liquidate and absorb

INV 4: Liquidation via repayAmount must never revert

INV 5: It must never be possible to repay more than a user's debt

INV 6: Absorb must only be called if position has bad debt

INV 7: discountedPrice must return USD price with 8 decimals

Privileged Functions

- none

Issue_05	Absorb logic can be used to drain the contract
Severity	**High**
Description	The absorb function allows addresses with the correct role to forgive any bad-debt and eventually decrease reserves or socialize it. There is currently no limit on how much debt can be decreased, as long as an account is considered as unhealthy, one can only remove the whole debt. **This can be abused by malicious liquidators as follows:** a) Create a position with collateral and borrow on behalf of it up to the threshold b) Wait 1 block to accrue interest such that the position becomes liquidatable c) Absorb the whole debt and withdraw the collateral The malicious liquidator now owns the borrowed amount as well as the collateral.
Recommendations	Consider implementing a stricter access control towards the absorb function.
Comments / Resolution	Resolved.

Issue_06	Liquidator can forcefully create bad-debt on own position and profit from it
Severity	**High**
Description	The liquidation discount is used to calculate the seizedAmount based on the repayAmount and vice-versa. It is simple decreasing the price of an asset and following these calculations: > repayAmount = seizedAmount * discountedPrice / 1e27 > seizedAmount = repayAmount * 1e27 / discountedPrice The goal here is to give the user a higher asset amount than the corresponding liquidated/repaid USD value, which incentivizes the liquidation execution. If an unreasonable high discount is used, this can result in not being able to properly liquidate positions. For example debt is meant to be repaid, it could result in a seizedAmount which is larger than the collateral, preventing the liquidation flow for this specific callpath. And on the other hand, the full collateral can just be taken and the repay amount will not cover the full debt. This not only happens if the liquidationDiscount is larger than LTV but also if the position is already above the LTV threshold and the liquidationDiscount is large enough to provoke this scenario. This can be trivially abused by liquidators to steal funds from the contract: Whenever a liquidation (for example) via the seizedAmount is executed, it is possible that the full supplied collateral is taken via the seizure and the full repayment is not happening. (Vice-versa it is also possible to liquidate via repayAmount, take the full collateral but not repay the full debt). To illustrate that, consider a simple example where Alice deposits

	100 DAI as collateral and borrows 75 FRAX, effectively meeting the LTV target of 75%. If now the discount is > 25%, let's say 27%, it is possible that Alice seizes the full 100 DAI on her own position while not fully repaying the 75 FRAX. a) Alice seizes 100 DAI -> repayAmount = 100 * 0.73 -> repayAmount = 73 FRAX b) Alice successfully seized 100 DAI while only repaying 73 FRAX and thus has profited 2 FRAX from this exploit.
Recommendations	Consider implementing a strong due-diligence check on the LIQUIDATOR_ROLE as well as ensuring that a reasonable liquidationDiscount corresponding to the LTV is chosen.
Comments / Resolution	Acknowledged.

Issue_07	Lack of confidence check within discountedPrice can result in incorrect price
Severity	**Medium**
Description	Whenever the discountedPrice is calculated, the oracle is consulted and meant to return the price of a user's collateral. Currently there is no confidence check applied which can result in over- or undercomputing the value of a collateral token, which then in turn can result in unallowed liquidations or the prevention of liquidations which should be allowed.
Recommendations	Consider incorporating the confidence within the discountedPrice function as well as incorporating a governance liquidation function in case the oracle does not work. The governance liquidation mechanism may require additional validation time.
Comments / Resolution	Acknowledged.

Issue_08	Unexpected side-effect in case of multi-iteration absorb can break the contract due to overflow in totalSupply
Severity	**Medium**
Description	The absorb function eventually sets the totalSupplyAssets amount to zero which means the contract comes into a state where totalSupply != 0 and totalSupplyAssets = 0. Essentially there are circulating shares with no corresponding value. If then a deposit happens in that scenario, it will effectively inflate the overall share amount due to the way how shares are computed. If this state is happening more than two times, the contract is essentially broken due to an overflow in share minting. **Illustrated:** > totalSupply = 100e18 > totalShares = 100e18 > absorb and empty full totalSupply > totalSupply = 0 > totalShares = 100e18 **First deposit after absorb** > Alice deposits 100e18 > assets * (totalSupply+1) / (totalAssets+1) > 100e18 * (100e18+1) / 1 > totalSupply = 1.000....1...000e40 + 100e18 > totalSuppliedAssets = 100e18 > absorb and empty full totalSupply > totalSupply = 0 > totalShares = 100000000000000000000200e18 (1.0002...000e40) **Second deposit after absorb**

	> Alice deposits 100e18 > assets * (totalSupply+1) / (totalAssets+1) > 100e18 * (100000000000000000000200e18+1) / (1) > received shares = 100000000000000000002000000000000000000000100e18 > totalSupply = 1.000...0000e60 If that happens again, the third deposit essentially reverts due to an overflow when shares are minted.
Recommendations	There is no trivial fix for this issue other than re-writing the erc4626 share calculation which could result in further side-effects. Since the likelihood of this issue happening in reality is very low plus the fact that we already recommended increasing the access control mechanism within the absorb function, it is reasonable to acknowledge this issue. In the scenario where an absorb would result in setting totalSupplyAssets to zero, governance can simply execute a deposit beforehand which prevents this scenario.
Comments / Resolution	Acknowledged.

Issue_09	Lack of collateral related storage adjustment during liquidation
Severity	**Medium**
Description	The liquidate function decreases the userCollateralSupplied[account][token] mapping by the seizedAmount but lacks to adjust both other collateral related variables: - collateralSupplied - userCollateralTokens While the non-removal of the token from the set does not expose any harm, the lack of collateralSupplied decrease will cause harm which is related to the maxSupply logic and the getTotalCollateral view-only return value.
Recommendations	Consider adjusting all collateral-related variables during the liquidation.
Comments / Resolution	Resolution 1: Failed resolution, collateralSupplied is not decreased. This has been fixed in the following commit: https://github.com/ambitfi/hyperdrive-contracts/commit/5dcdd7a388bb34ad95808b5c2830ac8b7f60619e please note that Bailsec did not validate this commit or any other commits after the commit provided in the resolution 1 table.

Issue_10	Liquidation may revert for zero-transfers
Severity	**Low**
Description	There are several tokens which revert upon zero transfers. This can become a problem during the liquidate function as it may possibly execute such zero transfers: *for (uint256 i; i < tokens.length; i++) {* *IERC20(tokens[i]).safeTransfer(msg.sender, seizedAmounts[i]);* *}*
Recommendations	Consider only executing a transfer if the amount is non-zero.
Comments / Resolution	Acknowledged.

Issue_11	Lack of close factor implementation
Severity	**Low**
Description	Currently it is possible to repay the full debt and seize the corresponding collateral based on the discount. This is a rather unconventional approach as it allows basically for full liquidation. Most lending protocols just allow for liquidating up to a certain amount of the debt such as 50%, which is known as the close factor.
Recommendations	Consider if this design choice has a very specific reason or if it makes more sense to implement a close factor. In the scenario where a close factor is implemented, we recommend a re-audit of both liquidation flows.
Comments / Resolution	Acknowledged.

Issue_12	Liquidation via repayAmount can result in loss for liquidator
Severity	**Low**
Description	The liquidation via repayAmount calculates the to received seizedAmount as follows: > seizedAmount = repayAmount * 1e27 / discountedPrice It is theoretically possible that seizedAmount becomes zero while the repayAmount is non-zero: **Illustrated:** > price = 1_000_000_000_000_000e8 > amount = 0.000000000000009e18 > scale everything to 27 > price = 100000000000000000000000000000000000000000000000 > amount = 900000000000000 > seizedAmount = amount * 1e27 / price 900000000000000* 1e27 / 100000000000000000000000000000000000000000000000 > 0.9 -> zero A similar issue can occur with repayAmount calculation. However, it must be noted that this will always round up. Additionally, there is always a general truncation when amounts are down-scaled when a descale is happening from 27 decimals to the target decimals, all digits are truncated to zero (until the target decimal is reached). Plus the last two digits from the USD return value will never be considered if asset decimals are 6.
Recommendations	Consider acknowledging this issue.

Comments / Resolution	Acknowledged.

MarketLib

The MarketLib contract is a library contract which is imported by the Market contract and facilitates all market related actions such as:

a) Borrow asset
b) Repay asset
c) Accrue interest
d) Health check
e) Reserve claiming
f) Flashloan
g) Liability calculation

Appendix: Liability Calculation

The borrow function allows users to borrow the asset in case they have sufficient collateral to stay healthy. Over the course of the borrowing time, borrowers are forced to pay interest on their debt. Since the interest is not per-borrow but rather on the overall borrowed amount, an ERC4626-like concept has been introduced which returns the real-time debt of each user, including the interest, which is usually in an constant-increasing manner due to the totalBorrowAssets increase on each accrue call (while totalBorrowShares remains consistent).

Whenever users borrow tokens, their internal share storage is increased, which is follows a similar concept to AAVE's debt token but resides in the contract storage:

> debt = userBorrowShares[account] * (totalBorrowAssets + accrued) / totalBorrowShares

If for example a user has initially borrowed 100e18 tokens which resulted in 100e18 shares and now interest is accrued, these 100e18 shares now result in a larger debt amount than 100e18.

Appendix: Borrow Flow

The borrow function allows users to borrow the asset which has been provided by the depositors, as long as the position is considered as healthy. Borrowing will increase a user's internal share allocation which represents the debt. The debt will increase over time and a user is required to repay the initially borrowed amount plus the debt before the full collateral can be withdrawn.

The flow for this function is as follows:

a) Accrue interest
b) Cache current liabilities
c) Invoke beforeBorrowHook
d) Calculate debt shares to receive for the borrowed amount
    i) > shares = amount * borrowShares / borrowAssets
e) Increase totalBorrowShares by shares
f) Increase totalBorrowAssets by borrowed amount
g) Increase userBorrowShares[account] by shares
h) Cache updated liabilities
i) Ensure position is healthy after borrow update
j) Determine optional borrowFee
k) Transfer optional borrowFee to borrowingFeeReceiver
l) Transfer borrowed amount - fee out
m) Invoke afterBorrowHook

Appendix: Repay Flow

The repay function allows users to repay their borrowed amount including any interest which has been accrued. A repayment will decrease the user's internal share allocation which reflects a decreased debt.

The flow for this function is as follows:

a) Ensure proper access control for position owner
b) Accrue interest
c) Cache current liabilities
d) Ensure liabilities are existent
e) Invoke beforeRepayHook
f) Clamp amount down to liabilities in case it is larger
g) Transfer repayAmount from caller in
h) Calculate share amount which is corresponding to repayAmount
    i) > shares = repayAmount * totalBorrowShares / totalBorrowAssets
i) Decrease totalBorrowAssets by amount
j) Decrease totalBorrowShares by shares
k) Decrease userBorrowShares[account] by shares
l) Invoke afterRepayHook

Appendix: Accrue

The accrual calculation works as follows:

a) Calculate the fractional rate (rate per second in WAD)
> *(1e18 + MathLib.wadDiv(rate, SECONDS_PER_YEAR_IN_WAD)).toUint128()*
> 1e18 + ratePerSecond (WAD)

b) Calculate the compounded rate (rate per seconds to the power of seconds)
> MathLib.wadPow(fractionalRate, elapsed).toUint128()
> fractionalRate ^ elapsed (WAD)

c) Apply the compounded rate on the previous index to calculate the percentual increase
> *MathLib.wadMul(previousIndex, compoundRate).toUint128()*
> previousIndex * compoundRate

d) Determine the accrued amount by multiplying the existing totalBorrowAssets value with the index delta:
> *MathLib.wadDiv(MathLib.wadMul(assets, index), previousIndex) - assets*
> (assets * index / previousIndex) – assets

Appendix Flashloan logic

The library exposes a simple flashloan functionality which allows a user to take out a flashloan and repay it in the same block without paying a fee.

Core Invariants:

INV 1: totalBorrowAssets must be increased over time due to interest

INV 2: Borrower must not be unhealthy after borrow execution

INV 3: toLiabilities must always be called with updated state

INV 4: collateral can only withdrawn if post-withdrawal position is still healthy

INV 5: Borrow must round received shares up

INV 6: Repay must round removed shares down

INV 7: Cannot repay more than liabilities

INV 8: isHealthy must always be called with updated liabilities as debt parameter

Privileged Functions

- none

Issue_13	Invalid oracle return values ignore supplied collateral
Severity	**High**
Description	Within the isHealthy function, a check is executed if the oracle return value is acceptable: *(USD price, IPriceOracle.Confidence confidence) = IPriceOracle(asset.priceOracle).getLatestPrice();* *if (confidence == IPriceOracle.Confidence.GOOD) {* *uint256 total = Math.mulDiv(* *MathLib.scale($.userCollateralSupplied[account][token], IERC20Metadata(token).decimals()),* *MathLib.scale(USD.unwrap(price), USD_DECIMALS),* *10 ** MathLib.SCALE* *);* If it is not acceptable, it will simply return all supplied collateral by the user without the supplied amount for the invalid oracle, resulting in a wrong computation of the overall collateral and thus an overall incorrect health check based on insufficient collateral consideration.
Recommendations	Consider reverting if the return value is not acceptable. However, it must be clear that this will prevent liquidations. Thus, we recommend implementing an additional governance liquidation function.

Comments / Resolution	Resolution 1: (Resolved), the isHealthy check was previously considered upon liquidations but this is now done via a different function. This means users will not get liquidated if the price is bad. However, this must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_14	Incorrect hook parameters for afterRepayHook and afterLiquidateHook
Severity	**High**
Description	Hooks are an essential instrument of this contract since they allow for arbitrary logic before and after executions. The outcome of the hook call is heavily based on the provided parameters. For example, the beforeRepayHook has the following parameters: account repayer amount liabilities As soon as these parameters are inaccurate, it will most likely happen that the outcome of the external call results in issues. Invalid parameters were found in the following hooks: a) afterRepayHook uses unupdated liabilities b) afterLiquidateHook uses unupdated liabilities Both hooks should use the correct liabilities after the execution.
Recommendations	Consider using the updated liability amount for both mentioned hooks.

Comments / Resolution	Partially resolved, afterLiquidateHook still uses the unupdated liability.

Issue_15	Lack of totalReserveAssets incorporation into borrow function
Severity	**Medium**
Description	Withdrawals can only be executed based on the availableLiquidity which deducts the totalReserveAssets value from the contract balance. Borrowing however does not do the same, allowing borrowers to borrow the full amount. This will prevent governance from claiming reserves.
Recommendations	Consider implementing the same safeguard towards the borrow function.
Comments / Resolution	Resolved.

Issue_16	Asset is inherently assumed to be in USD
Severity	**Medium**
Description	The whole architecture relies on the fact that the asset is denominated in USD, which can be examined within the isHealthy function. This will not work if the asset is not a stablecoin and will also give invalid collateralization rates if the stablecoin is depegged, as it will then show a higher debt than the user in reality has.
Recommendations	Consider only adding stablecoins as the asset.
Comments / Resolution	Acknowledged.

Issue_17	Lack of borrowing threshold
Severity	**Medium**
Description	Usually, lending protocols do not allow to borrow up to the exact threshold as this then possibly triggers immediate liquidation in the next block. The same counts for withdrawals of the collateral. A safety margin is usually in place which prevents something like that. However, the current protocol design does not incorporate this, which means users can be immediately liquidated if they don't exercise caution.
Recommendations	Consider either placing a warning on the frontend or implementing such an additional safeguard. Validation of this safeguard is necessary with additional time.
Comments / Resolution	Resolution 1: Failed resolution, an additional FIELDS_LIQUIDATION_LTV field has been added which is tied to the liquidation LTV. It is assumed that the liquidation LTV is higher than the maxLTV to ensure a proper threshold. However, the withdraw function within the Market contract calls isHealthy with FIELDS_LIQUIDATION_LTV which means a user can still accidentally withdraw up to the liquidation threshold, essentially no safeguard is applied here (unlike during the borrow function). Moreover, as mentioned this must be fully audited and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update. The client has conducted a change in the following commit: https://github.com/ambitfi/hyperdrive-contracts/commit/98cf37d82afd3f91a2645af8a62142c1bb43e7ad this has not been audited by Bailsec but is an attempt to fix the failed resolution. This status is subject to update.

Issue_18	Truncation before aggregation can result in position being marked as unhealthy
Severity	Medium
Description	The isHealthy function loops over all tokens and calculates the total value as follows: userCollateralSupplied * tokenPrice / 1e27 Afterwards, this is descaled to the asset decimals and aggregated in the limit: limit += MathLib.mul(asset.maxLTV, MathLib.descale(total, decimals)); The descaling will scale the value from 1e27 to asset decimals. This will always truncate digits. If for example multiple iterations are executed and everytime, digits are truncated, it can result in the limit being below the debt while it would actually in reality be above if no truncation has happened. Quick example: userCollateralSupplied 1 = 100000099999999999999999 userCollateralSupplied 2 = 100000099999999999999999 price = 1e8 LTV = 1e18 The result if both values would be aggregated together without truncation: 2.000002e24 The result if both values are aggregated together including truncation: 2e24

	If now the limit is 2.000001e24, the position is considered as unhealthy while it should be healthy. The root-cause is the truncation before aggregation.
Recommendations	Consider implementing a borrow limit which does not allow to borrow up to the threshold. This will prevent such corner cases most of the time. Moreover, consider descaling the final aggregated amount instead of each value on its own.
Comments / Resolution	Resolution 1: (Resolved), scaling is now handled differently while the oracle return value is with 27 decimals. This will prevent most truncation scenarios. Moreover, the isHealthy was refactored and the new isLiquidation function was added. However, as mentioned this must be fully audited (due to the refactoring) and formally verified with all possible edge-cases in order to mark as resolved in an effort to ensure no side-effects are introduced due to the new implementation. This status is subject to update.

Issue_19	Potential OOG within isHealthy loop
Severity	**Low**
Description	The isHealthy function loops over all tokens in the userCollateralTokens set in the scenario where the set becomes unreasonably large, this loop will consume excessive gas which can result in a function revert due to the “out of gas” issue.
Recommendations	Consider ensuring that not too many collateral tokens are added by governance.
Comments / Resolution	Acknowledged.

Issue_20	Withdrawals can be griefed by borrowing
Severity	**Low**
Description	Currently, there is no reserve which prevents borrowing a certain percentage of the deposited asset amount. If borrowing does not cost a fee, an annoying griefing attack can be used to borrow whenever a user wants to withdraw which thus results in a revert.
Recommendations	Consider either implementing a reserve or ensuring that the borrowingFee is never 0.
Comments / Resolution	Acknowledged.

Issue_21	Violation of checks-effects-interactions pattern during repay
Severity	**Low**
Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes). - repay - liquidate
Recommendations	Consider implementing a reentrancy guard for all external functions.
Comments / Resolution	Resolved.

Issue_22	BorrowingFeeMax bypass via aggregator
Severity	**Low**
Description	An aggregator which is built on top of the Markets can be used to batch multiple single borrow requests together, using the borrowingFeeMax safeguard to bypass paying the fee for each single position.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_23	Potential loss during repay
Severity	**Low**
Description	The repay function calculates the to removed amount of shares down which is in itself the correct way to do: > shares = repayAmount * totalBorrowShares / totalBorrowAssets It is however theoretically possible that the totalBorrowAssets value grows exponentially large which could result in a loss during the repayment of a small amount because the divisor is so high which results in zero shares (depending in the seize of repayAmount)
Recommendations	Consider ensuring that shares != 0.
Comments / Resolution	Resolved, it has to be noted that it can still occur that truncation is happening.

Market

The Market contract is the heart of the architecture and exposes all directly callable functions. The actual logic of most functions is outsourced to the above listed library contracts.

The following core functionalities are present:

a) Users can deposit and withdraw the asset with the goal to accrue interest paid by borrowers
b) Users can deposit collateral which can then be used as overcollateralized position to borrow the asset
c) Users can withdraw collateral as long as this does not make the position unhealthy
d) Users can borrow the asset which will expose them to paying interest on it
e) Users can repay the asset including all accrued interest
f) Users can take a fee-free flashloan

Appendix: Asset Deposit Flow

The Market contract exposes an ERC4626-like system which allows for depositing the asset via the deposit or mint function while receiving the corresponding amount of shares for it. The calculation follows the standard ERC4626 calculation:

> shares = assetAmount * totalShares / totalAssets

> assets = shareAmount * totalAssets / totalShares

The only difference is that the totalAssets function is not fetched via balanceOf but as follows:

> totalSupplyAssets + (accrued - (reserveFee * accrued)

While totalSupplyAssets is simply increased with each deposit as well as during the accrual to reflect the interest which is paid back to the lenders.

Appendix: Asset Withdrawal Flow

The market contract exposes an ERC4626-like system which allows for withdrawing the asset token via the withdraw or redeem function while receiving the corresponding asset amount for the provided share amount. The calculation follows the standard ERC4626 calculation. The only difference is the same as described above.

Appendix: Operator Assignment

Similar as in other contracts developed by Hyperdrive, the owner of a position can assign an operator via the setOperator function which essentially grants the operator full privileges over the owner's position. In order for this to be valid, governance must mark the operator additionally as allowed operator via the setApprovedOperator function

Appendix: Value determination

As already explained within the deposit and withdrawal flow calculation, the value determination of the asset's shares follow an ERC4626-like concept. Instead of increasing balanceOf, the totalSupplyAsset variable is increased upon each accrue call which simply reflects the interest which is due to be paid by borrowers to lenders. The increase of this variable will be then reflected in the totalAssets function, essentially increasing the value of each share, reflecting a fair interest distribution among all lenders:

> amount = shareAmount * totalAssets / totalShareAmount

Core Invariants:

INV 1: Accrue must be called before any interaction

INV 2: toLiabilities must always be called with updated state

INV 3: Users must only be able to withdraw balanceOf - totalReserveAssets

INV 4: Every action which decreases the health factor must execute an isHealthy check after the state change

INV 5: toLiabilities must always round up

INV 6: Asset deposits must never exceed maxSupply

INV 7: Users cannot withdraw more than availableLiquidity

Privileged Functions

- setCollateralAsset
- setInterestRateModel
- setMaxSupply
- setBorrowingFee
- setReserveFee
- setHooks
- setApprovedOperator

Issue_24	Governance Issue: Full control over configuration
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, governance can a) Update the oracle address b) Change the LTV c) Introduce malicious hooks
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_25	Interest can be prevented due to lack of access control for beforeResume
Severity	**High**
Description	The beforeResume function is meant to be called after the contract has been paused and sets lastUpdate to block.timestamp which ensures no interest is accrued within the paused time. This function lacks an access control mechanism, making it publicly callable for everyone which allows to prevent interest accumulation.
Recommendations	Consider adding an access control mechanism to this function.
Comments / Resolution	Resolved.

Issue_26	Reentrancy attack during asset deposit via ERC777 token allows for manipulating the exchange rate
Severity	**High**
Description	The standard OZ ERC4626 logic first transfers the token in and then mints the shares. This is done in an effort to prevent reentrancy-attacks if the token is an ERC777 token since the hook is triggered before the balance increase, which is a valid state. In the scenario where a token is used with a hook after the balance change, this methodology is invalid. This is however not the issue here. The problem is that the totalSupplyAssets variable is increased before the transfer which means even if the hook is invoked before the balance increase, this is already an invalid state which results in an inflated exchange rate which can then be used by a malicious user to reenter into the withdraw function and benefit from that faulty state.

	Illustrated: **Status Quo:** >totalSupply = 100e18 >totalSupplyAssets = 100e18 >ER = 1 **Attacker deposits 100e18 tokens (normal deposit)** > totalSupply = 200e18 > totalSupplyAssets = 200e18 > ER = 1 > attackerShares = 100e18 **Attacker deposits 100e18 tokens to trigger attack + reenter into withdrawal** > receivedShares = 100e18 > totalSupplyAssets = 300e18 > ER = 1.5 > reenter during transfer in; call withdraw > withdraws 100e18 shares > gets 150e18 tokens out > receives 100e18 shares > totalSupplyAssets = 150e18 > totalShares = 200e18 > attacker withdraws 100e18 shares > receives 75e18 tokens > attacker deposited 200e18 and received 225e18 out
Recommendations	Consider adding a reentrancy guard not only to the deposit and mint but **all** external callable functions in the contract.
Comments / Resolution	Resolved.

Issue_27	Potential asset withdraw frontrun before absorb
Severity	**Medium**
Description	The absorb function potentially decreases totalSupplyAssets which inherently decreases the exchange rate of the ERC4626 vault for the asset. A smart user can detect that some accounts have accrued bad-debt and withdraw before the absorb function is called which means all other lenders will bear now the loss while the user was clever enough to bypass it.
Recommendations	Consider if it makes sense to implement a grace period for withdrawing or if this issue can be accepted. It is also possible to call absorb via a flashbot to prevent frontrunning.
Comments / Resolution	Acknowledged.

Issue_28	Vault inflation attack in case collateral = asset
Severity	**Medium**
Description	During the normal business logic, it is not possible to trigger the vault inflation attack since the totalSupplyAssets variable is only increased upon deposits and upon accrue whenever funds are borrowed. This means the contract will never be in the state to increase totalSupplyAssets by a large amount without any existing shares (because nothing can be borrowed if nobody has deposited). However, in case the asset is at the same time also allowed as collateral, there is a sophisticated way to execute the vault inflation attack, by depositing 1 wei of the asset to increase totalSupplyAssets by 1 and minting 1 share, subsequently followed by adding a larger amount of the asset as collateral and borrowing it. Using this methodology it is possible to increase

	totalSupplyAssets due to the interest application, which is possible because borrowing large sums is allowed even without a regular large asset deposit. While the vault inflation attack cannot be fully executed towards a zero truncation due to the following check: *require(shares > 0, ZeroAmountNotAllowed());* It is still possible to inflate the divisor in such a way to truncate one digit to zero: 19e18 * 2 / (20e18+1) = 1.899 = 1 (truncated)
Recommendations	Consider adding a check which prevents the asset from being added as a collateral token.
Comments / Resolution	Resolved.

Issue_29	Change of borrowingFeeMax and borrowingFee can result in unexpected loss for borrowers
Severity	**Medium**
Description	Governance can change both aforementioned variables which will have an inherent impact on a borrower who has created a borrow request just before these values are changed (if stuck in the mempool). This will inherently result in a loss for the borrower.
Recommendations	Consider implementing a maxBorrowingFee parameter into the borrow function OR notifying the community via all communication channels before such a change.
Comments / Resolution	Acknowledged.

Issue_30	Liquidations are disallowed if contract is paused
Severity	**Medium**
Description	Currently, liquidation is not possible if the contract is paused. This can result in bad-debt situations due to the lack of immediate action.
Recommendations	Consider implementing an admin liquidation mechanism which allows for liquidation during paused states.
Comments / Resolution	Resolved, an admin liquidation mechanism has been implemented.

Issue_31	Side-effects when using asset as collateral token
Severity	**Medium**
Description	There are several side-effects when the asset is used as collateral token, including but not limited to: - Manipulation of utilization rate (totalSupplyAssets is not increased but the collateral amount can be used for borrow purposes) - Collateral may not be withdrawn even if position is healthy because it is borrowed out - Can withdraw asset which is other user's collateral - Advanced inflation attack
Recommendations	Consider adding a check which prevents adding the asset as collateral token.
Comments / Resolution	Resolved.

Issue_32	General lack of reentrancy guards
Severity	**Medium**
Description	Multiple functions do not follow the CEI pattern and at the same time it is possible that ERC777 tokens are used, additionally, the withdrawCollateralETH function transfers out native ETH before the isHealthy check. While we could not find a definitive exploit from that design, we still consider it as a potential vulnerability for more sophisticated exploit ideas. This issue is rated as medium due to the fact that the contract multiple times violates the CEI and exposes several different entry functions which allows for multiple potential attack paths.
Recommendations	Consider implementing a reentrancy guard on all functions.
Comments / Resolution	Resolved.

Issue_33	Lack of reasonable validation for reserveFee
Severity	**Medium**
Description	The setReserveFee function allows for setting the reserveFee . Currently, there is no validation which ensures that the setting will result in a reasonable value, this can result in an underflow revert within the accrueLiabilities function.
Recommendations	Consider adding a validation for this variable.
Comments / Resolution	Resolved.

Issue_34	Lack of zero amount check during repay and borrow
Severity	**Low**
Description	Both aforementioned functions can be called with amount = 0 which is basically a no-operation. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider ensuring proper validation measurement which prevent such calls. This should be implemented for all other user-callable functions as well.
Comments / Resolution	Resolved.

Issue_35	Liquidation execution pre-oracle update
Severity	**Low**
Description	Depending on which oracle is being used, it is possible to frontrun the aggregator price updated with a liquidation call. For example, if the collateral is worth 1 USD in reality but the Chainlink oracle still shows the old value which could be 0.9 USD, while at the same time the price update for the Chainlink oracle is in the mempool, a liquidator can frontrun this update and liquidate a position, receiving a larger seizedAmount. Obviously this does not only count for the collateral price but also for the asset price. A similar possibility exists in case the LTV is near 100% and the oracle update moves the price a lot. This can be abused to steal from the protocol.

Recommendations	Consider keeping this scenario in mind as well as implementing a reasonable LTV.
Comments / Resolution	Acknowledged.

Issue_36	Off-by-one error in _deposit
Severity	**Low**
Description	Within the _deposit function, the following off-by-one error is exposed: *require($.totalSupplyAssets + assets < $.maxSupply, MaximumSupplyExceeded($.maxSupply));* This will essentially not allow reaching the expected maxSupply.
Recommendations	Consider changing the condition to <=.
Comments / Resolution	Resolved.

Issue_37	Lack of accrue during withdrawCollateral/ETH
Severity	**Low**
Description	At first glance, it does not appear to be an issue that the collateral withdrawal does not trigger the accrue function. However, on further investigation there is an edge-case which is related to the change of the IRM. If withdrawCollateral/ETH is called, the previewLiabilities function calculates the liabilities based on the current IRM. If now the IRM is changed after the withdrawCollateral function has been triggered and the rate is now increased, this means the liabilities from the previous preview are incorrect as they are now in reality higher (due to the increased interest rate). If accrue would have been called, the lastUpdate variable would have been updated and the new IRM logic would not be applied in hindsight. This can result in a user executing a legitimate withdrawal, followed by an IRM change which increases liabilities, making the “healthy” position “unhealthy”.
Recommendations	Consider preventing such an issue by simply calling accrue at the beginning of the collateral withdrawal.
Comments / Resolution	Resolved.

Issue_38	Interest is already accrued while not being paid back
Severity	**Informational**
Description	A similar issue is existing in most lending protocols: Interest is already accrued while it has not been paid back by borrowers, which essentially allows lenders to already withdraw any interest which has not yet been paid back. This increases the chance of bad-debt.
Recommendations	Since it is a known downside from the design of interest-based lending protocols, there is no trivial fix for this issue.
Comments / Resolution	Acknowledged.

Issue_39	PausableUpgradeable is not initialized
Severity	**Informational**
Description	The PausableUpgradeable contract is inherited but not initialized, this is against best practices.
Recommendations	Consider initializing the PausableUpgradeable contract.
Comments / Resolution	Resolved.

Issue_40	Transfer-tax incompatibility
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.

InterestModel

DynamicInterestRateModel

The DynamicInterestRateModel is a simple external interest calculation contract which can be plugged in and out of the Market contract. It simply calculates the interest rate based on the utilization rate and baseRate, kink, slope0 and slope1

Appendix: Utilization Calculation

The rate calculation which is explained below is based on the current utilization rate of the contract which is usually determined as follows:

> totalLiabilities / totalAssets

The following extra cases are present:

a) totalAssets = 0 and totalLiabilities = 0
   > zero rate

b) totalAssets = 0 and totalLiabilities != 0
   > 1e18 rate

c) totalAssets != 0 and totalLiabilities = 0
   > zero rate

Appendix: Rate Calculation

The previewRate function handles the calculation of the rate based on the utilization rate which was calculated as explained above. There are two different scenarios:

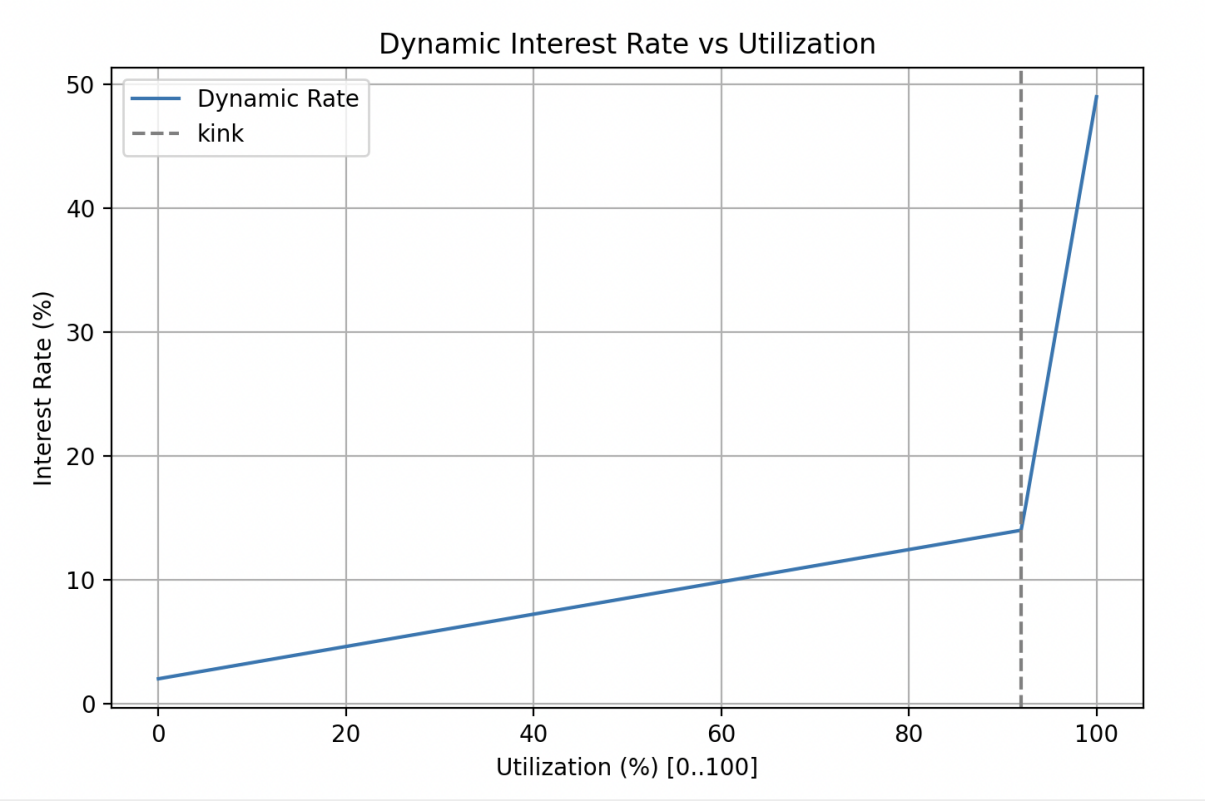
a) Utilization below kink:
   > baseRate + ((slope0 * utilization) / kink)

b) Utilization above/equal kink:
   > baseRate + (slope0 + (slope1 * ((utilization - kink) / (1e18 - kink))

Based on the provided parameters:

a) baseRate = 0.02e18
b) kink = 0.92e18
c) slope0 = 0.12e18
d) slope1 = 0.35e18

The following graph can be used to illustrate the interest rate based on the current utilization rate:



Core Invariants:

INV 1: kink must never be set to 1e18

INV 2: kink must never be set to 0

INV 3: fractionalRate must always represent rate per second

INV 4: Utilization must never be larger than 1e18

INV 5: CompoundRate must never be below 1e18

INV 6: FractionalRate must never be below 1e18

Privileged Functions

- none

Issue_41	Division by zero error for certain kink settings
Severity	**Medium**
Description	Within the above appendix, we explained the mathematical formulas for calculating the rate: c) Utilization below kink: > baseRate + ((slope0 * utilization) / kink) d) Utilization above/equal kink: > baseRate + (slope0 + (slope1 * ((utilization - kink) / (1e18 - kink)) a) will revert due to division by zero if kink = 0 b) Will revert due to division by zero if kink = 1e18
Recommendations	Consider ensuring that kink is none of these values.
Comments / Resolution	Resolved.

Hooks

CollateralHooksLib

The CollateralHooksLib is a simple library contract which facilitates the execution of hooks for collateral related actions such as supplyCollateral/ETH and withdrawCollateral/ETH

It exposes the following hooks:

a) beforeSupply
b) afterSupply
c) beforeWithdraw
d) afterWithdraw

Each execution allows multiple hooks with corresponding options. For example, an asset can have three hooks while only one hook is triggered during beforeSupply. This is all settable by governance during the addOrUpdateAsset function.

The flow of hook execution is as follows:

a) beforeXHook: In case hooks array contains a valid element, invoke beforeX
b) beforeX: Loops over each valid hook and invokes ensureBeforeX in case the hook has the corresponding flag
c) shouldCallBeforeX: Checks whether the hook has the corresponding flag set
d) ensureBeforeX: Invokes the beforeX function on the corresponding hook, validates return selector and returns context data

Core Invariants:

INV 1: The hook return value must match the corresponding selector

Privileged Functions

- none

Issue_42	Potential out of bound array access in rare scenario
Severity	**Low**
Description	Both “before” hooks create return data which consists out of an array of data based on the length of the hooks. This data is then decoded again during each “after” hook and a loop is executed over the same length of hooks which accesses the corresponding data. If there is any exotic hook execution during the “before” hook which calls the addOrUpdateAsset function to change the hook logic including increasing the amount of hooks (for whatever reasons), it will then loop over the increased hooks length during the “after” hook, which will inherently attempt to access an out of bound element in the encoded data. This has the reason that the initially decoded data is only from the previous hooks size while the loop now is executed over a larger size.
Recommendations	Consider keeping this in mind in such a hook development scenario.
Comments / Resolution	Acknowledged.

Issue_43	Potential issue if only “after” flag is set
Severity	**Informational**
Description	If a hook sets only the “AFTER” bit, never the “BEFORE” bit. The “before” loop never writes data[i], leaving data[i] empty (""). When the “after” loop runs, it tries data[i] as input to the after function and passes this as an argument into the after hook call (which is essentially empty). If now the after hook expects corresponding context, it will not work. Note that this is not an issue which stems from the code itself but rather from incorrect settings.
Recommendations	Consider keeping that in mind when setting the hook configuration
Comments / Resolution	Acknowledged.

Issue_44	Potential call to address(0) hook
Severity	**Informational**
Description	It is theoretically possible that a hook with address(0) but the corresponding flags is set which would then attempt to call the corresponding hook function on address(0), resulting in a revert.
Recommendations	Consider ensuring that hooks are always set correctly.
Comments / Resolution	Resolved.

MarketHooksLib

The MarketHooksLib exposes similar logic to the CollateralHooksLib and handles hooks for market-related actions, such as:

a) beforeDeposit
b) afterDeposit
c) beforeWithdraw
d) afterWithdraw
e) beforeBorrow
f) afterBorrow
g) beforeRepayHook
h) afterRepayHook
i) beforeLiquidateHook
j) afterLiquidateHook

Core Invariants:

INV 1: The hook return value must match the corresponding selector

Privileged Functions

- none

Issue_45	Potential out of bound array access in rare scenario
Severity	**Low**
Description	Both “before” hooks create return data which consists out of an array of data based on the length of the hooks. This data is then decoded again during each “after” hook and a loop is executed over the same length of hooks which accesses the corresponding data. If there is any exotic hook execution during the “before” hook which calls the addOrUpdateAsset function to change the hook logic including increasing the hook size (for whatever reasons), it will then loop over the increased hooks length during the “after” hook, which will inherently attempt to access an out of bound element in the encoded data. This has the reason that the initially decoded data is only from the previous hooks size while the loop now is executed over a larger size.
Recommendations	Consider keeping this in mind in such a hook development scenario.
Comments / Resolution	Acknowledged.

Issue_46	Potential issue if only “after” flag is set
Severity	**Informational**
Description	If a hook sets only the “AFTER” bit, never the “BEFORE” bit. The “before” loop never writes data[i], leaving data[i] empty (""). When the “after” loop runs, it tries data[i] as input to the after function and passes this as an argument into the after hook call (which is essentially empty). If now the after hook expects corresponding context, it will not work. Note that this is not an issue which stems from the code itself but rather from incorrect settings.
Recommendations	Consider keeping that in mind when setting the hook configuration
Comments / Resolution	Acknowledged.

Issue_47	Potential call to address(0) hook
Severity	**Informational**
Description	It is theoretically possible that a hook with address(0) but the corresponding flags are set which would then attempt to call the corresponding hook function on address(0), resulting in a revert.
Recommendations	Consider ensuring that hooks are always set correctly.
Comments / Resolution	Resolved.