

STATE MIND

Fluid

30-10-2023 - 29-12-2023



1. Project Brief		5
2. Finding Severity breakdown		8
3. Summary of findings		9
4. Conclusion		9
5. Findings report		10
Critical	The rate from LendingRewardRateModel is returned with the incorrect number of decimals	10
	BigMathVault library contains math error that causes debtFactor to be calculated completely wrong	11
	Storing invalid debt factor value in tick	13
High	Incorrect supplyExchangePrice calculation	15
	Incorrect extract of inFrom_ address	16
	Impossible to change the borrowing mode of a user in the Liquidity Layer	17
	DOS of iTokens markets	18
	Wrong slippage checks	19
	Possible reward overpayment	20
	Possible bad debt in StETHQueue	21
Incorrect work of UniV3CheckFallbackCLRSOracle in the first mode	21	
Medium	Rounding errors in calcTokenData	22
	Incorrect exchange prices extraction	23
	Excessive expandPercent leads to underflow in withdrawal limit	24

Medium

Sanity checks for utilization	25
Unsustainable economy of iToken	25
Mixed compound rewards of iToken	26
Unclear shares_ amount in redeemWithSignature method	27
DOS via reward model	28
Incorrect MAX_RATE in the rewards model	29
Tick IDs overflowing	29
Wrong debt calculation of liquidated positions	30
Logical error in StETHQueue.queue leading to revert	31
Incorrect rounding in StETHQueue.claim	31
Stale data in chainlink oracles	32
DoS of uniV3OracleImpl	33

Informational

Proxy redundant functional	33
Optimization of unstructured storage	34
Redundant memory variable	34
Non-optimal if condition	34
Incorrect set of WithdrawalLimit	35
Unused BigMath functions	36
Little gas optimizations	36
Storage update of exchange prices	37
Double calculations	37
Potential Misconfiguration of RateData	38
Typos in comments	38
Possible DOS with rate manipulation	39

Informational

Using same slot for proxy admin and governance	39
Zero address check for _revenueCollector	39
Redundant function and storage value overwriting	40
Checking callbackData_ length	40
Auth role potential impact	40
Additional checks for Liquidity's token configs	41
Redundant check	41
Sanity check for uint to int conversion	41
Code duplicate	42
Wrong slot numbers in comments	42
Unused imports	42
Redundant input variable	42
Redundant type conversion	43
Memory to calldata gas optimization	43
Redundant variable in scope	43
Code duplication	43
Incorrect BigInt sizes in comments	44
Double calculation of vault_ address	44
Unnecessary cast to payable address	44
Unrestricted maxLTV in StETHQueue	45
Optimization of rate calculation	45
Strict sanity check of TWAP periods	45
Negative tick cumulative processing	46
Revert with equal deltas	46
Incorrect logic of to_ in operate method of Vault	47

Informational

User can bypass the check in flashLoanMultiple	47
maxFlashLoan calculates an incorrect amount due to lack of subtracting of userBorrow_ from borrow limit.	47
Unused library function	47

1. Project Brief



Title	Description
Client	Instadapp
Project name	Fluid
Timeline	30-10-2023 - 29-12-2023
Initial commit	a324cc2faccf3947c712f2c29ea0affa0620cbe5
Final commit	f5a07116967103946791dff1fbafa71e0a60828

Short Overview

Fluid Protocol is a DeFi lending and borrowing platform offering high loan-to-value ratios, innovative liquidation mechanisms, and features like smart debt and collateral.

At the base of Fluid lies the Liquidity Layer, which serves as the foundation upon which other protocols can be built. This layer serves as a central hub where liquidity from all protocols is consolidated. The Liquidity Layer allows protocols to deposit, withdraw, borrow, and payback. End-users interact with the protocols, which in turn interact with the Liquidity layer.

Currently Fluid includes lending, vault, stETH, flashloan protocols, robust oracle system.

Project Scope

The audit covered the following files:

- [uniV3TWAP3Chainlink1Oracle.sol](#)
- [chainlink3HopOracle.sol](#)
- [chainlink2HopOracle.sol](#)
- [chainlink3HopOracleImpl.sol](#)
- [uniV3OracleImpl.sol](#)
- [FullMath.sol](#)
- [liquiditySlotsLink.sol](#)
- [bigMath.sol](#)
- [proxy.sol](#)
- [error.sol](#)
- [constantVariables.sol](#)
- [main.sol](#)
- [main.sol](#)
- [error.sol](#)
- [lendingRewardsRateModel.sol](#)
- [iTokenEIP2612Deposits.sol](#)
- [variables.sol](#)
- [vaultFactory.sol](#)
- [ERC721.sol](#)
- [errorTypes.sol](#)
- [proxy.sol](#)
- [main.sol](#)
- [errorTypes.sol](#)
- [fluidOracle.sol](#)
- [chainlink1HopOracle.sol](#)
- [wstETHOracleImpl.sol](#)
- [chainlinkOracleImpl.sol](#)
- [errorTypes.sol](#)
- [tickMath.sol](#)
- [errorTypes.sol](#)
- [error.sol](#)
- [main.sol](#)
- [helpers.sol](#)
- [events.sol](#)
- [errorTypes.sol](#)
- [helpers.sol](#)
- [main.sol](#)
- [iTokenPermit2Deposits.sol](#)
- [lendingFactory.sol](#)
- [IVaultFactory.sol](#)
- [main.sol](#)
- [events.sol](#)
- [error.sol](#)
- [structs.sol](#)
- [errorTypes.sol](#)
- [structs.sol](#)
- [wstETHChainlink2HopOracle.sol](#)
- [wstETHOracle.sol](#)
- [error.sol](#)
- [chainlink2HopOracleImpl.sol](#)
- [TickMath.sol](#)
- [storageRead.sol](#)
- [liquidityCalcs.sol](#)
- [events.sol](#)
- [errorTypes.sol](#)
- [events.sol](#)
- [IVault.sol](#)
- [variables.sol](#)
- [variables.sol](#)
- [errorTypes.sol](#)
- [iTokenNativeUnderlying.sol](#)
- [events.sol](#)
- [error.sol](#)
- [vaultT1Logic.sol](#)
- [structs.sol](#)
- [variables.sol](#)
- [errorTypes.sol](#)
- [events.sol](#)


 [main.sol](#)

 [proxy.sol](#)

 [helpers.sol](#)

 [structs.sol](#)

 [structs.sol](#)

 [dummyImpl.sol](#)

 [events.sol](#)

 [variables.sol](#)

 [error.sol](#)

2. 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 to be transferred to any party.
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 of funds.
Informational	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.

3. Summary of findings



Severity	# of Findings
Critical	3 (3 fixed, 0 acknowledged)
High	8 (7 fixed, 1 acknowledged)
Medium	15 (9 fixed, 6 acknowledged)
Informational	40 (33 fixed, 7 acknowledged)
Total	66 (52 fixed, 14 acknowledged)

4. Conclusion



During the audit of the codebase, 66 issues were found in total:

- 3 critical severity issues (3 fixed)
- 8 high severity issues (7 fixed, 1 acknowledged)
- 15 medium severity issues (9 fixed, 6 acknowledged)
- 40 informational severity issues (33 fixed, 7 acknowledged)

The final reviewed commit is `f5a07116967103946791dffd1fbafa71e0a60828`

5. Findings report



CRITICAL-01	The rate from LendingRewardRateModel is returned with the incorrect number of decimals	Fixed at <u>df1195</u>
-------------	--	------------------------

Description

The rate returned from the `getRate` function of the `LendingRewardRateModel` is scaled up by `1e2`, which results in 14 decimals instead of the intended 12.

Due to this precision adjustment, all following calculations are performed in 14 decimals, so as return value of `getRate`.

```
uint256 precisionAdjustment_ = RATE_PRECISION - INPUT_PARAMS_PERCENT_PRECISION;
```

`CONSTANT1` is also calculated with wrong precision

```
CONSTANT1 = rateData_.rateAtTVLZero * RATE_PRECISION;
```

The rate is later used in `_calculateNewTokenExchangePrice`, but as the unchecked block assumes `// rewardsRate is in 1e12`, `totalReturnInPercent_` increases unexpectedly significant. This later enhances `newTokenExchangePrice_`. This exchange price then passes through `_updateRates` straight to `_executeDeposit` and `_executeWithdraw` functions. Finally, overpricing leads to wrong shares calculation in such way that allows pool draining.

PoC was handed to client.

Recommendation

```
-uint256 precisionAdjustment_ = RATE_PRECISION - INPUT_PARAMS_PERCENT_PRECISION;  
+uint256 precisionAdjustment_ = RATE_PRECISION / INPUT_PARAMS_PERCENT_PRECISION;
```

```
-CONSTANT1 = rateData_.rateAtTVLZero * RATE_PRECISION;  
+CONSTANT1 = rateData_.rateAtTVLZero * precisionAdjustment_;
```

CRITICAL-
02

BigMathVault library contains math error that causes debtFactor to be calculated completely wrong

Fixed at
[9573a2](#)

Description

At the end of liquidation, branch's debt factor updates with `mulDivBigNumber` function, but result is wrong. At [Line 498](#) `branch_.debtFactor` is initialized and its value is set to 50-bit number. According to protocol logic function `mulDivBigNumber()` should return 50-bit value, but it could exceed 50 bit. This leads to an underestimation of the liquidated position.

```
// branchData_ >> 116 contains branch debtFactor_ in BigNumber format and could exceed 50 bit. So operation & could lead to wrong conversion from BigNumber
```

```
memoryVars_.minimaDebtFactor = (branchData_ >> 116) & X50;
```

```
positionRawDebt_ = positionRawDebt_.mulDivNormal(  
  memoryVars_.minimaDebtFactor, // <-- wrongly trimmed BigNumber  
  memoryVars_.connectionFactor // <-- Here we assume that the connectionFactor value equals to initial branch debtFactor value  
);
```

In this example `memoryVars_.minimaDebtFactor` will be wrongly converted and maybe way less than `memoryVars_.connectionFactor` which will result in position reset.

PoC: [test/foundry/vaultT1/vault /PoC_debt_factor_overflow.t.sol](#)

Incorrect value returns due to a mistake in return value [calculation](#).

```
result =  
  (_resultNumerator << COEFFICIENT_SIZE_DEBT_FACTOR) + // should be EXPONENT_SIZE_DEBT_FACTOR  
  (bigNumber & EXPONENT_MAX_DEBT_FACTOR) +  
  diff -  
  PRECISION; // + exponent
```

Also, the optimization of the diff calculation is incorrect. In both cases, diff must be greater by 1.

```
uint256 diff = (_resultNumerator > TWO_POWER_COEFFICIENT_PLUS_PRECISION_MINUS_1_MINUS_1)  
  ? COEFFICIENT_PLUS_PRECISION_MINUS_1 // In this case diff should be 99, not 98  
  : (_resultNumerator > TWO_POWER_COEFFICIENT_PLUS_PRECISION_MINUS_2_MINUS_1)  
  ? COEFFICIENT_PLUS_PRECISION_MINUS_2 // In this case diff should be 98, not 97  
  : BigMathMinified.mostSignificantBit(_resultNumerator);
```

Recommendation

We recommend changing the constant in the result calculation

```
uint256 diff = (_resultNumerator > TWO_POWER_COEFFICIENT_PLUS_PRECISION_MINUS_1_MINUS_1)  
  - ? COEFFICIENT_PLUS_PRECISION_MINUS_1  
  + ? COEFFICIENT_PLUS_PRECISION_MINUS_1 + 1  
  : (_resultNumerator > TWO_POWER_COEFFICIENT_PLUS_PRECISION_MINUS_2_MINUS_1)  
  - ? COEFFICIENT_PLUS_PRECISION_MINUS_2  
  + ? COEFFICIENT_PLUS_PRECISION_MINUS_2 + 1  
  : BigMathMinified.mostSignificantBit(_resultNumerator);
```

```
result =  
- (_resultNumerator << COEFFICIENT_SIZE_DEBT_FACTOR)  
+ (_resultNumerator << EXPONENT_SIZE_DEBT_FACTOR) +  
  (bigNumber & EXPONENT_MAX_DEBT_FACTOR) +  
  diff -  
  PRECISION; // + exponent
```

Description

During liquidation protocol passes through all ticks that have debt, accumulates debt and liquidates it, updating the **debtFactor_** value after every iteration at [Line 825](#). If some tick is not liquidated, then at [Lines 617 – 627](#) its state is changed and **branch_.debtFactor** is stored as tick's **debt factor**. It is used to recalculate position after liquidation in function [fetchLatestPosition\(\)](#):

```
if (((tickData_ >> 1) & X24) == positionTickId_) {
  // fetching from tick data itself
  // ...

  memoryVars_.connectionFactor = (tickData_ >> 56) & X50; // <-- here connectionFactor is tick's debt factor saved in
liquidation
} else {
  {
    // Fetching tick's liquidation data. One variable contains data of 3 IDs. Tick Id mapping is starting from 1.
    // ...

    memoryVars_.connectionFactor = (tickLiquidationData_ >> 31) & X50; // <-- here connectionFactor is tick's debt
factor saved in liquidation
  }
}

// ...
// ...
// ...

positionRawDebt_ = positionRawDebt_.mulDivNormal( // <-- position debt is calculated based on initial tick's debt
factor and final branch's debt factor
  memoryVars_.minimaDebtFactor,
  memoryVars_.connectionFactor // <-- here we assume that tick's branch was not merged
);
```

Because liquidation occurs iteratively ticks in one branch should have different initial debt factors. At [Line 627](#) **branch_.debtFactor** is saved, which is only updated (we don't account for merge) at [Line 801](#) when liquidation is over. Due to this logic positions with ticks, that were liquidated later in the cycle, will be calculated incorrectly and will underestimate debt and collateral.

PoC: [test/foundry/vaultT1/vault/PoC_initial_tick_debt_factor.t.sol](#)

Recommendation

It is recommended to save **branch_.debtFactor** in ticks multiplying it by current **debtFactor_** or updating **branch_.debtFactor** after every iteration. E.g

```
// ...
```

```
tickData[currentData_.tick] =
```

```
  1 |
```

```
  (((temp2_ >> 1) & X24) << 1) |
```

```
  (branch_.id << 26) |
```

```
  (branch_.debtFactor.mulDivBigNumber(debtFactor_) << 56);
```

```
// ...
```

Description

At [Lines 109 - 110](#) **supplyExchangePrice_increase** is calculated and then added to **supplyExchangePrice**. **supplyExchangePrice** is a number of normal tokens per one raw token. But in code numerator is in normal tokens and denominator is also in normal format:

```
temp_ = (temp_ * EXCHANGE_PRICES_PRECISION) / // <- here temp_ is supply increase in normal tokens
        (tokenData_.supplyRawInterest * tokenData_.supplyExchangePrice); // <- multiplication raw tokens by price gives us
total number of supply tokens in normal format

tokenData_.supplyExchangePrice += temp_;
```

So, actually, **temp_** is not **supplyExchangePrice_increase**, it is just increase of normal tokens. This causes underestimation of accrued interests and results in less profit for lenders.

Recommendation

It is recommended to count **supplyExchangePrice_increase** based on this formula:

```
temp_ = temp_ / tokenData_.supplyRawInterest;

tokenData_.supplyExchangePrice += temp_; // <- now temp_ is in correct price format
```

Description

At [Line 561](#) `inFrom_` address is extracted from `callbackData_`, but `bytes` memory location is not taken into account. In memory `bytes` consists of two parts: first 32 bytes with data length, and second one is actual data. So when pad is added to `callbackData_` in `assembly`, it begins not from actual data, but from slot with length and extraction goes wrong:

```
assembly {
  inFrom_ := mload(add(callbackData_, sub(mload(callbackData_), 20)))
          // ^- here pad is calculated and added to length slot, not to actual memory slot with data
}
```

Also with `abi.encode()` address variable will be padded with zeroes on the left to result in total 32 bytes. So, 32 bytes should be subtracted from length, because `assembly` expects address with leading zeros. When it is simple 20 bytes, conversion will be also incorrect.

Such extraction may lead to incorrect checks passing and detecting DEX-protocols.

Recommendation

It is recommended to modify extraction of address from `callbackData_` to take into account memory location of `bytes` and conversion in `assembly`:

```
assembly {
  inFrom_ := mload(add(add(callbackData_, 32), sub(mload(callbackData_), 32)))
}
```

Description

To change the mode of a user we need to call the `updateUserBorrowConfigs` method of **AdminModule** of Liquidity Layer. Let's look at the part where the mode changes, e.g. [Lines739-771](#).

```
// .....
totalBorrowRawInterest_ = tokenData_.borrowRawInterest;
totalBorrowInterestFree_ = tokenData_.borrowInterestFree;

// read current user borrowing & borrow limit values
borrowingConversion_ = (userBorrowData_ >> 1) & X64; // here borrowingConversion_ = user borrow amount
borrowingConversion_ =
    (borrowingConversion_ >> DEFAULT_EXPONENT_SIZE) <<
    (borrowingConversion_ & DEFAULT_EXPONENT_MASK);

// .....

if (userBorrowData_ & 1 == 0 && userBorrowConfigs_[i].mode == 1) {
    // ....

    // decreasing interest free total borrow
    totalBorrowInterestFree_ -= borrowingConversion_; // total = total - user borrow

    // problem here -----^

    // .....

} else if (userBorrowData_ & 1 == 1 && userBorrowConfigs_[i].mode == 0) {

    // .....

    // decreasing raw (with interest) borrow
    totalBorrowRawInterest_ -= borrowingConversion_; // total = total - user borrow raw

    // problem here -----^

    // .....
}
```

borrowingConversion_ is a borrowed amount of a user whose mode is changed. **totalBorrowRawInterest_** is the total borrow amount of users whose **mode** is **1** and **totalBorrowInterestFree_** is the total borrow amount of users whose **mode** is **0**.

Due to calculations in UserModule **borrowingConversion_** is stored rounded up([link](#)). **totalBorrowRawInterest_** and **totalBorrowInterestFree_** are stored rounded down([link1](#), [link2](#)).

This leads to arithmetic underflow if there is only one protocol with a particular mode, let it be mode = 0. Then **totalBorrowInterestFree_** should be equal to **borrowingConversion_**, cause there is only one protocol with this mode. But **totalBorrowInterestFree_** is rounded down and **borrowingConversion_** is rounded up so **totalBorrowInterestFree_ - borrowingConversion_ < 0**. That's a place where underflow happens.

So if we once have 2 different protocols with different modes it is impossible to make all protocols have the same mode cause while we change them one by one the situation where only one protocol has a particular mode and others have another mode will occur. E.g. switch all modes to become 0 or 1.

Recommendation

We recommend adding a check here([link1](#), [link2](#)) that the subtracted is less than the reduced like

```
if (totalBorrowInterestFree_ < borrowingConversion_) {
  totalBorrowInterestFree_ = 0;
} else {
  totalBorrowInterestFree_ -= borrowingConversion_;
}
```

and

```
if (totalBorrowRawInterest_ < borrowingConversion_) {
  totalBorrowRawInterest_ = 0;
} else {
  totalBorrowRawInterest_ -= borrowingConversion_;
}
```

Also consider rounding up the total borrowed amounts here([link1](#), [link2](#)).

HIGH-04

DOS of iTokens markets

Fixed at [72c32e](#)

Description

Function `createToken()` is open and anyone can create iToken for certain **asset** if this underlying is configured in **Liquidity**. Address of **iToken** is determined only by **asset** address. So if **iToken** is already deployed, there would be no possibility to deploy **iToken** with the same **asset**, even of a different type. Caller also provides type for **iToken**. He can choose **NativeUnderlying** for **asset** he would like to DOS. This will most likely pass the check in **Liquidity**, because at [Lines 163 - 165](#) **NATIVE_TOKEN_ADDRESS** will be chosen for further check. **NativeUnderlying** type of **iToken** blocks any deposits and withdrawals for regular tokens in underlying.

```
bytes32 liquidityExchangePricesSlot_ = LiquiditySlotsLink.calculateMappingStorageSlot(
  LiquiditySlotsLink.LIQUIDITY_EXCHANGE_PRICES_MAPPING_SLOT,
  iTokenType_ == ITokenType.NativeUnderlying
  ? NATIVE_TOKEN_ADDRESS
  : asset_
); // <- here any token type can be choosed with any asset_, so anyone can use ITokenType.NativeUnderlying
if (LIQUIDITY.readFromStorage(liquidityExchangePricesSlot_) == 0) {
  // ^- this check could be passed because NATIVE_TOKEN_ADDRESS would be used in liquidityExchangePricesSlot_
  // instead of asset_ address
  revert FluidLendingError(ErrorTypes.LendingFactory__LiquidityNotConfigured);
}
```

Functions `_executeDeposit()` and `_executeWithdraw()` are overridden and call `withdraw()` and `deposit()` of underlying tokens, which most tokens don't have.

Recommendation

It is recommended to restrict `msg.sender` of `createToken()` and use singleton pattern for **iToken** of **NativeUnderlying** type. With `msg.sender` restriction `salt` can be calculated from `asset_` address and **ITokenType** to allow creation different **iToken** types for the same `asset_`.

Description

In `mint()` function there is a slippage check:

```
if (assets_ < minAmountOut_) {  
    revert FluidLendingError(ErrorTypes.iToken__MinAmountOut);  
}
```

But it is logically wrong, because user indicates amount of share he wants to mints and should indicate max amount of **assets_** he would like to spend during mint. So this check is meaningless and opens possibility to manipulations. E.g. someone sees **mint** transaction, then makes big withdraw in current block to inflate **rate** and **tokenExchangePrice_** in the next block. This action will decrease **supplyRawAmount** in **Liquidity**, so **supplyExchangePrice** will be higher, at the same time **total shares** will decrease and **rewardsRate** will be also higher resulting in higher **tokenExchangePrice** in **iToken**.

Same issue:

- Function `withdrawWithSignature()`: there should be check **shares_** doesn't exceed **maxAmountOut_**
- Function `redeemWithSignature()`: there should be check **assets_** are not less than **minAmountOut_**
- Function `mintWithSignature()`: there should be check **assets_** doesn't exceed **maxAmountOut_**
- Function `mintWithSignature()`: there should be check **assets_** doesn't exceed **maxAmountOut_**
- Function `mintETH()`: there should be check **assets_** doesn't exceed **maxAmountOut_**
- Function `withdrawETH()`: there should be check **shares_** doesn't exceed **maxAmountOut_**
- Function `withdrawWithSignatureETH()`: there should be check **shares_** doesn't exceed **maxAmountOut_**

Recommendation

It is recommended to fix slippage checks to avoid possible manipulations during deposits and withdrawals.

Description

The `totalReturnInPercent_factor` calculates reward term as multiplication of `rewardsRate_` and `timeElapsed`.

```
totalReturnInPercent_ =
  (rewardsRate_ * (block.timestamp - _lastUpdateTimestamp)) /
  _SECONDS_PER_YEAR;
```

The `rewardsRate_` will be `> 0` only within `[START_TIME, END_TIME]` timeframe.

The `_lastUpdateTimestamp` updates if:

1. Admin adds rewards model via `updateRewards`.
2. Admin adds rewards via `fundRewards` after model is set.
3. Anyone withdraws/deposits while rewards are active i.e. `address(rewardsRateModel_) != address(0)`;
4. Anyone calls `updateRates`.

```
-----|-----|-----|-----
updateRewards--^   ^--START   ^--END
```

If arbitrary users don't make new deposits/withdrawals (or `updateRates`) until the model's `START_TIME` is reached, the `_lastUpdateTimestamp` variable will be outdated, the actual duration of rewards will be longer. Which means the protocol users that already have shares, are incentivised to not interact with the protocol to get extra rewards. (More likely to happen in small markets).

If reward amounts calculated precisely, even small overpayment might lead to DOS for latest withdraw if not reimbursed by the protocol.

Recommendation

We recommend funding rewards right before the `START_TIME` or calculating the rewards only within the specified timeframe `[START_TIME, END_TIME]`.

HIGH-07

Possible bad debt in **StETHQueue**

Acknowledged

Description

The **StETHQueue** contract allows users to borrow ETH against Lido's **WithdrawalQueue** NFT withdrawal requests, with the max amount borrowed based on a percentage (**maxLTV**) of their withdrawing **stETH** amount. The borrowed ETH comes from the **Liquidity** contract, and users are expected to repay the borrowed amount plus accrued interest.

An issue arises when the repay amount exceeds the **stETHAmount**. In such cases, users lack both the capability and the incentive to repay the debt, leading to an irrecoverable bad debt. The **StETHQueue** contract accumulated debt in the **Liquidity** contract without a possibility of recovering it with "bad" withdrawal requests.

Recommendation

It is recommended to implement a liquidation mechanism. This mechanism should allow to claim requests when the ratio of **repayAmount** to **stETHAmount** exceeds a predetermined threshold.

Client's comments

For this we:

- a) manage the risk for Lido slashing via the maxLTV, with some leftover risk that we accept
- b) if a user doesn't claim and the increase through borrow rate leads to outgrowing the collateral value the method **claim()** is open to execute by anyone and we can trigger it ourselves.
- c) The stETH protocol contract is upgradeable to deal with any unforeseen cases.

The incentive to repay the bad debt is for the owner to avoid it. We do however really not expect a depeg of stETH to happen to the extent where it would go beyond the maxLTV limit (or a slashing so drastic). As far as we know this is a widely accepted risk in the industry by now and we decided to also accept this risk. In the event that it would happen after all, we accept that it might take a bit longer to first implement a solution (e.g. similar to the one proposed) and upgrade the contract before being able to resolve the situation.

HIGH-08

Incorrect work of **UniV3CheckFallbackCLRSOracle** in the first mode

Fixed at [a413c2](#)

Description

In the first mode, when **_RATE_SOURCE == 1**, there should be no checks for **exchangeRate_**. But in condition at [Lines 68 - 71](#) there is no **return** statement, so execution continues. It will always revert on delta check at [Lines 90 - 92](#) because value **checkRate_** won't be set and will equal zero.

Recommendation

It is recommended to return **exchangeRate_** as it was calculated when **UniV3CheckFallbackCLRSOracle** is configured on the first mode.

Description

In `calcTokenData`, performing a division in the middle of a calculation can result in smaller `supplyExchangePrice` than expected. For instance, 100 rounds of supply-borrow result in `supplyExchangePrice` of 1020325352672 when division is performed at the end, and 1020325352650 if not.

Recommendation

We recommend rearranging operations to always perform divisions as late as possible, while also ensuring the intermediate results do not overflow. e.g.

```
-temp_ = (tokenData_.borrowExchangePrice * temp_ * secondsSinceLastUpdate_) / (SECONDS_PER_YEAR *
FOUR_DECIMALS);
-tokenData_.borrowExchangePrice += temp_;
+ temp_ = (tokenData_.borrowExchangePrice * temp_ * secondsSinceLastUpdate_);
+ tokenData_.borrowExchangePrice += temp_ / (SECONDS_PER_YEAR * FOUR_DECIMALS);
...
-temp_ = (temp_ * EXCHANGE_PRICES_PRECISION) / (tokenData_.supplyRawInterest * tokenData_.supplyExchangePrice
);
+temp_ = (temp_ * EXCHANGE_PRICES_PRECISION) / (tokenData_.supplyRawInterest * tokenData_.supplyExchangePrice
) / (FOUR_DECIMALS * SECONDS_PER_YEAR);
```

Description

The **updateTokenConfigs** function extracts the exchange prices from the storage **_exchangePricesAndConfig** via shifting the value.

```
supplyExchangePrice_ = (exchangePricesAndConfig_ >> 97) & X64;  
borrowExchangePrice_ = (exchangePricesAndConfig_ >> 161) & X64;
```

Although exchange prices are packed in **_exchangePricesAndConfig** with offsets

BITS_EXCHANGE_PRICES_SUPPLY_EXCHANGE_PRICE = 91 and

BITS_EXCHANGE_PRICES_BORROW_EXCHANGE_PRICE = 155 respectively, offsets in the code are different.

Due to the wrong offsets **supplyExchangePrice_** will not contain the first 6 bits of actual value, moreover it will have the first 6 bits of the actual **borrowExchangePrice**. Following the same logic, **borrowExchangePrice_** will not have 6 bits of the saved value and also might become greater due to reading the **supply ratio flag** and 5 bits of the **supply ratio** value.

After the storage read, it may occur that one of the extracted values will be 0, which may lead to the resetting of both exchange prices. Or on contrary, it will not reset values when it has to.

```
if (supplyExchangePrice_ > 0 && borrowExchangePrice_ > 0) {  
    ...  
    // updates exchange prices  
    ...  
} else {  
    supplyExchangePrice_ = EXCHANGE_PRICES_PRECISION;  
    borrowExchangePrice_ = EXCHANGE_PRICES_PRECISION;  
}
```

Recommendation

We recommend using implemented constant values as offsets.

Description

The function **updateUserSupplyConfigs** lacks a sanity check to ensure that the **expandPercent** value does not exceed **FOUR_DECIMALS**, though there is a check to confirm that **expandPercent** fits within 14 bits (which limits it to 16383). Exceeding **FOUR_DECIMALS** will cause an underflow in the function **calcWithdrawalLimitBeforeOperate**, which next will result in a DoS for user withdrawals.

```
function calcWithdrawalLimitBeforeOperate(
    uint256 userSupplyData_,
    uint256 userSupply_
) internal view returns (uint256 currentWithdrawalLimit_) {
    ...

    uint256 expandPercent_ = (userSupplyData_ >> 162) & X14;
    uint256 maxWithdrawableLimit_ = (expandPercent_ * userSupply_) / FOUR_DECIMALS;
    // ^- can be higher than userSupply_

    ...

    uint256 minWithdrawalLimit_;
    unchecked {
        // subtraction can not underflow as maxWithdrawableLimit_ is a percentage amount of userSupply_
        minWithdrawalLimit_ = userSupply_ - maxWithdrawableLimit_;
        // ^- however, it can underflow
    }
    if (minWithdrawalLimit_ > currentWithdrawalLimit_) {
        currentWithdrawalLimit_ = minWithdrawalLimit_; // <-- will put underflowed value
    }
}
```

The result will happen **_supplyOrWithdraw** function due to condition:

```
if (amount_ < 0 && userSupply_ < newWithdrawalLimit_) { // <-- will revert here due to underflow
    revert FluidLiquidityError(ErrorTypes.UserModule__WithdrawalLimitReached);
}
```

Recommendation

We recommend adding a sanity check in **calcWithdrawalLimitBeforeOperate** to ensure that **expandPercent** does not exceed **FOUR_DECIMALS**.

MEDIUM-04	Sanity checks for utilization	Fixed at 336754
<p>Description</p> <p>Based on the layout in <code>_exchangePricesAndConfig</code> mapping utilization occupies 14 bits. During calculations in <code>_operate()</code> function at Line 387 utilization is stored as <code>uint256</code> and potentially could exceed 16386 ($2^{14} - 1$). This could happen when all lent tokens are borrowed and borrow rate is very high – its maximum value could be up to 650%. So during saving utilization value in storage at Lines 443-453, overwrite could happen and values of other variables may not be saved correctly.</p> <p>Recommendation</p> <p>It is recommended to add sanity after calculating utilization value and limit it.</p>		

MEDIUM-05	Unsustainable economy of iToken	Acknowledged
<p>Description</p> <p>The iToken contract accepts user deposits, which are then transferred to the Liquidity contract. The iToken contract holds a stake in the Liquidity contract, funded by user deposits and accessible for withdrawal by users.</p> <p>iToken operates on a unique supply rate model, incentivizing users to deposit. The growth of iToken shares is based on the growth of liquidity supply exchange price and its rewards rate model, which calculates rewards based on the current TVL. The contract should maintain the invariant that every user can withdraw their entire share at any time. However, under the active rewards rate model, a user's withdrawal amount may exceed their share in Liquidity. This discrepancy creates a gap between the estimated TVL in <code>totalAssets()</code> and the actual balance in Liquidity, violating the invariant. Consequently, this gap could prevent users from withdrawing their assets.</p> <p>Although the <code>updateRewards</code> function is intended to fund the rewards, it does not guarantee users' ability to withdraw their funds (even without rewards).</p> <p>Recommendation</p> <p>We recommend establishing a separate balance within the contract specifically for rewards. This balance should be replenished through <code>updateRewards</code> and accessible for withdrawal via a separate function. This approach would mitigate the gap issue, ensuring more reliable withdrawals for users.</p> <p>Client's comments</p> <p>We will implement off-chain automation for this that will fund rewards whenever there is a difference in <code>totalAssets()</code> vs. <code>liquidityBalance()</code>. Depositors can thus be sure they will get the rewards and withdrawals will be possible at all times, even though there is admittedly some trust required there. Not perfect but good enough as rewards are only used at the initial launch of an iToken.</p> <p>Note that we added a cap for rewards rate at 25% to ensure rewards could not be "inflated" as a simple additional measure in Commit1 and Commit2.</p>		

Description

The exchange price of iToken is computed as follows:

- $rewardsRate = rewardsRateModel(totalSupply \cdot prevTokenExchangePrice)$
- $totalReturnInPercent = \frac{rewardsRate \cdot timeElapsed}{secondsPerYear} + \frac{liquiditySupplyExchangePrice - prevLiquiditySupplyExchangePrice}{prevLiquiditySupplyExchangePrice}$
- $tokenExchangePrice = (1 + totalReturnInPercent) \cdot prevTokenExchangePrice$

Therefore, iToken's rewards, including the growth in the liquidity supply exchange price, accumulate in every contract interaction, representing a compound reward. This implies that the rewards are not limited solely by the rewards rate model.

Recommendation

We recommend separating rewards and growth of liquidity supply exchange price, so rewards can be capped by rewards rate model.

Client's comments

So depending on the rate, in some cases, it is beneficial for lenders to not have anyone interact whereas in some cases it would be beneficial to trigger the compounding to earn more rewards. Rewards will only go on for a few months after launching a new iToken. Absolute precision is not required here.

Compounding every second at 10% for 3 months will result in the user getting 2.5315% rather than getting 2.5% and it's fine. As long as there is no exploit possible in a sort of looping then this difference is ok.

Description

In order to redeem with signature sender must provide exact **shares_** amount to pass checks in **permit** method otherwise it reverts.

However it is unclear before real execution how much **shares_** will be passed to **permit** function.

Let's closer look here

```
function redeemWithSignature(
  uint256 shares_,
  address receiver_,
  address owner_,
  uint256 minAmountOut_,
  uint256 deadline_,
  bytes calldata signature_
) external virtual nonReentrant returns (uint256 assets_) {
  // ...
  assets_ = previewRedeem(shares_);
  // ...
  shares_ = _executeWithdraw(assets_, receiver_, owner_);
  // ...
  (uint8 v_, bytes32 r_, bytes32 s_) = _splitSignature(signature_);
  // ...
  // spender = msg.sender
  permit(owner_, msg.sender, shares_, deadline_, v_, r_, s_);
  // ...
}
```

shares_ are recalculated from **_executeWithdraw** method and can differ from provided shares.

The reason is following. Provided **shares_** are converted to **assets_** via `previewRedeem` method which itself uses such a [formula](#):

```
return shares_.mulDivDown(tokenExchangePrice_, _EXCHANGE_PRICES_PRECISION);
// ^-- rounded down
```

Later calculated **assets_** converted to new **shares_** in `_executeWithdraw` method using this [formula](#):

```
sharesBurned_ = assets_.mulDivUp(_EXCHANGE_PRICES_PRECISION, tokenExchangePrice_);
// ^-- rounded up
```

Due to different rounding types the resulting **shares_** can differ from provided so permit will likely revert.

Recommendation

We recommend adding one more argument storing the exact amount of shares user wants to spend and check that resulting **shares_** equal to the requested or do not exceed it like

```

function redeemWithSignature(
  uint256 sharesToPermit_
  uint256 shares_,
  // ...
) external virtual nonReentrant returns (uint256 assets_) {
  // ..
  shares_ = _executeWithdraw(assets_, receiver_, owner_);
  if (shares_ > sharesToPermit_) {
    revert "YOUR_FluidError here";
  }
  // ...
  permit(owner_, msg.sender, sharesToPermit_, deadline_, v_, r_, s_);

  _spendAllowance(owner_, msg.sender, shares_);
}

```

MEDIUM-08

DOS via reward model

Fixed at [86827b](#)

Description

In the `getRate` function, reward model checks if the calculated rate is less than `MAX_RATE` and reverts if it's not.

```

if (rate_ > MAX_RATE) {
  revert FluidLendingError(ErrorTypes.LendingRewardsRateModel__MaxRate);
}

```

If the underlying function allows such rates that might reach the maximum value, this line might lead to the DOS of all the linked `iToken` contracts.

Arbitrary users can not withdraw their assets as the call to reward contract happens before `_burn` until tvl reaches the value `getRate(tvl) > MAX_RATE`.

```

uint256 tokenExchangePrice_ = _withdrawFromLiquidity(assets_, receiver_);

tokenExchangePrice_ = _updateRates(tokenExchangePrice_, false);
// calls to the `getRate` --^

sharesBurned_ = assets_.mulDivUp(_EXCHANGE_PRICES_PRECISION, tokenExchangePrice_);

if (sharesBurned_ == 0) {
  revert FluidLendingError(ErrorTypes.iToken__RoundingError);
}
// reduces totalSupply() --v
_burn(owner_, sharesBurned_);

```

Recommendation

We recommend returning the `MAX_RATE` value if rate exceeds the maximum.

```

if (rate_ > MAX_RATE) {
  rate_ = MAX_RATE;
}

```

MEDIUM-09	Incorrect MAX_RATE in the rewards model	Fixed at 86827b
<p>Description</p> <p>The MAX_RATE is defined as constant in the lending reward model contract as 20000%, while comments state the maximum value must be 200%. The high amount for MAX_RATE might never be reached.</p> <pre>uint256 internal constant MAX_RATE = 20_000 * RATE_PRECISION;</pre> <p>Recommendation</p> <p>We recommend reducing the actual maximum reward rate.</p>		

MEDIUM-10	Tick IDs overflowing	Acknowledged
<p>Description</p> <p>The variables.sol contract contains positionData and tickData mappings in which 24 bits are allocated for users tick IDs. Considering that tick IDs gradually <u>increase</u> with each liquidation, 24 bits $\sim 10^7$ may not be enough for the long-term operation of the protocol (about 10^7 liquidations). In particular, DOS may occur due to overwriting of ticks: operate function has such an <u>if-else</u> construction:</p> <pre>if (((temp_ & 1) == 1) (((temp_ >> 1) & X24) > o_.tickId)) { //<-- invalid tickID due to overflow // User got liquidated //... } else { // User didn't got liquidated //... }</pre> <p>Recommendation</p> <p>We recommend increasing the number of bits for TickID.</p> <p>Client's comments</p> <p>I believe the overflow will never happen as let's consider a scenario where the same tick is getting liquidated every hour. That means collateral price went down, the tick became available for liquidation, collateral price went up, the user(s) created a new position at the same tick, and the collateral price went down again. So for a tick to get liquidated again this 4-step process needs to happen again and again. Considering that it happens every 1 hour then it'll take about this many seconds to overflow: $(2^{24} - 1) * 60 = 1006632900$ secs = 31.9201198630137 years. So the collateral price needs to revolve around that tick for ~ 32 years in order for tickId to overflow.</p>		

Description

Position's debt consists of two parts: real users' debt and dust debt:

$$debt = users_debt + dust_debt$$

During liquidations whole debt is liquidated, so both parts of it decrease.

E.g. x is percent of debt after liquidation, so:

$$debt * \frac{x}{100} = users_debt * \frac{x}{100} + dust_debt * \frac{x}{100}$$

In **operate()** function at Lines 130 - 163 debt of liquidated position is calculated. In function **fetchLatestPosition()**

positionRawDebt_ is whole debt, including user's and dust. Then user's debt is calculated at Line 156. But dust debt wasn't recalculated in **fetchLatestPosition()**, so user debt will be:

$$user_debt_liq = debt * \frac{x}{100} - dust_debt$$

Instead of:

$$user_debt_liq = debt * \frac{x}{100} - dust_debt * \frac{x}{100}$$

This leads to underestimation of user's debt amount.

Recommendation

It is recommended to recalculate dust debt due to liquidation the same way as total debt is recalculated in **fetchLatestPosition ()**.

Client's comments

For user, the actual debt is **debt - dustDebt**. Users also don't pay any extra interest on dustDebt. When the liquidation happens it happens of a tick so it contains **totalDebt = debt + dustDebt**, on liquidation totalDebt starts to get liquidated but when the user fetches his position after liquidation the user gets **debt = debtAfterLiquidation - dustDebt**. So all the liquidation is deducted from their actual debt. With dustDebt user's liquidation starts to happen about ~0.15% early.

Description

The `StETHQueue.queue` function contains a logical error related to the dividing on withdrawal batches, which leads to revert when `stETHAmount_ % MAX_STETH_WITHDRAWAL_AMOUNT == 0`.

```
...

bool lastAmountExact_;
uint256 fullAmountsLength_ = stETHAmount_ / MAX_STETH_WITHDRAWAL_AMOUNT;
unchecked {
    lastAmountExact_ = stETHAmount_ % MAX_STETH_WITHDRAWAL_AMOUNT == 0;
    amounts_ = new uint256[(fullAmountsLength_ + (lastAmountExact_ ? 0 : 1))];
    // ^- will have length of fullAmountsLength_ if lastAmountExact_
}

for (uint256 i; i < fullAmountsLength_; ) {
    amounts_[i] = MAX_STETH_WITHDRAWAL_AMOUNT;

    unchecked {
        ++i;
    }
}

if (lastAmountExact_) {
    amounts_[fullAmountsLength_] = MAX_STETH_WITHDRAWAL_AMOUNT;
    // ^- will revert due to index error
} ...

...
```

In the case of `lastAmountExact_ == true`, the `amounts` is allocated correctly after the for loop. The if-branch after is incorrect and will lead to a revert due to an index error.

Recommendation

It is recommended to remove the if-branch when `lastAmountExact_ == true`.

Description

The `StETHQueue.claim` function contains an issue with the rounding method used in calculating the `repayAmount_`. The current implementation rounds down due to simple division:

```
repayAmount_ = (claim_.borrowAmountRaw * _getLiquidityExchangePrice()) / EXCHANGE_PRICES_PRECISION;
```

This rounding down can result in the `repayAmount_` being slightly less than what it should be, leading to an incomplete repayment of the debt.

Recommendation

It is recommended to adjust the calculation of `repayAmount_` to round up instead of down.

Description

It's better to check oracles for stale data because some data can be outdated.

Same can be applied for redstone implementation.

Also, If the project will be launched on L2's where can be Sequencer, it's better to add a check for it.

Recommendation

We recommend adding checks for stale data.

Client's comments

For mainnet we decided to accept the risk regarding stale data similar to other major DeFi protocols (as far as we know) e.g. Compound, Aave. Note that we aim to use Oracles that implement the check against another Oracle as implemented wherever possible, or even the UniV3 TWAP Oracle for any token pair with sufficient liquidity.

For L2's / other networks we will implement stale data / other necessary checks as suggested, added a part to the docs in: <https://github.com/Instadapp/fluid-contracts/pull/222/commits/736f42cf4942398ec7d6dad328a3818085853179>

Description

In **uniV3OracleImpl**, the current price is compared to TWAPs with different periods. Each TWAP has its own delta percent. The closer the TWAP edge is to the current price, the smaller the delta. It is checked in function `_checkTWAPDelta()`:

```
...

maxDelta_ = (price_ * maxDelta_) / OracleUtils.HUNDRED_PERCENT_DELTA_SCALER;
if (exchangeRate_ > (price_ + maxDelta_) || exchangeRate_ < (price_ - maxDelta_)) {
    // Uniswap last price is NOT within the delta
    revert FluidOracleError(ErrorTypes.UniV3Oracle__InvalidPrice);
}

...
```

The current price is calculated based on the last saved **tickCumulative** value in **observations**. The attacker couldn't manipulate the price to be changed in oracle, because it would revert due to comparisons with old TWAPs, but he could DOS oracle with these reverts. He can change the tick's value in every block by the minimal delta value. Direction of price change is not important for the attack, but malicious actor can buy **token0** in one block, and then sell it in another to mitigate losses from arbitrage.

Recommendation

It is recommended not to use the current price in a pool as **exchangeRate_**, instead use TWAP with a longer period and compare its value with other TWAPs, being accurate with oracle checkpoints, deltas, and collateral factor. Also **try - catch** block returning **0** can be used instead of revert and processed in oracles.

Client's comments

Using a longer TWAP for the current price can be done through the config passed in with the constructor, so no code changes are needed (current price = first TWAP interval price). However, we are not sure if using a longer TWAP as current price is the best solution here. The attack of delaying liquidation without reaching bad debt by selling token0 in one block and buying token1 in the next would be a "just having fun" attack, but even that does not come cheap. With such a long TWAP window we would not be able to cause sudden dumps, causing liquidations to be lagging, potentially causing bad debt. For example, if the price falls by 5 or 10% in the last 30 min, the TWAP will stop working considering it as manipulation but that's an actual price decrease. UniV3 oracle will be checked against an Oracle like Chainlink or Redstone.

Description

The **receive** function in the proxy contract contains a redundant functional in case of **msg.sig > 0**. This check is intended to differentiate between empty and non-empty calldata. However, in EVM, the **receive** function is used for plain Ether transfers, which inherently have empty calldata. This means **msg.sig** in the context of **receive** will always be zero.

Recommendation

We recommend removing redundant functional for gas saving in deployment and in plain transfers.

INFORMATIONAL-02	Optimization of unstructured storage	Fixed at c69413
<p>Description</p> <p>The current method of interacting with the arbitrary storage slots via structures is more gas consuming than the assembly sload & sstore.</p> <p>E.g.:</p> <ul style="list-style-type: none"> • <code>_getAddressSlot</code> • <code>_getSigsSlot</code> • storage read via structure in <code>_getGovernanceAddr</code> <p>Recommendation</p> <p>We recommend using the same implementation as Lido's UnstructuredStorage (used in StorageRead library), which consumes less gas on both load and store.</p>		

INFORMATIONAL-03	Redundant memory variable	Fixed at 4f16d3
<p>Description</p> <p>In the GovernanceModule contract's functions <code>updateAuths</code> & <code>updateGuardians</code>, memory variable <code>setStatus_</code> is created only to be used once immediately after declaration.</p> <p>Recommendation</p> <p>We recommend removing redundant variables to consume less gas.</p>		

INFORMATIONAL-04	Non-optimal if condition	Fixed at 14219e
<p>Description</p> <p>In the <code>_operate</code>, during the supply/borrow ratio calculations the first ifs can be optimized by replacing two checks (both uint) with check of the already calculated sum.</p> <p>Also, the if body sets to 0 the same variable that condition checked to be 0.</p> <pre data-bbox="178 1780 1921 2047"> temp_ = InterestFree + temp3_; - if (temp3_ == 0 && InterestFree == 0) { + if (temp_ == 0) { temp3_ = 0; } </pre> <p>Recommendation</p> <p>We recommend optimizing conditions.</p>		

Description

In function `calcWithdrawalLimitBeforeOperate()`, **WithdrawalLimit** is calculated. There is inconsistency between calculations, docs and comments in code. Based on docs and comments **expandPercent** is maximum number of tokens that can be withdrawn when full **expandDuration** is passed. So withdraw amount increases linearly over the **expandDuration** period. At [Line 194](#) `currentWithdrawalLimit_` is calculated based on `lastWithdrawalLimit_` and `expandedWithdrawableAmount_`.

Consider a case:

```

user_supply = 6_000_000

expandPercent = 1000 // 10%
expandDuration = 200 // 200 seconds

// First we deposit

newWithdrawalLimit_ = 0 // it is 0 because it is a first user interaction

user_supply = 6_000_000

// calcWithdrawalLimitAfterOperate()
newWithdrawalLimit_ = user_supply - ((user_supply * expandPercent) / 10_000) = 5_400_000;

// Then we withdraw after 100 seconds

// calcWithdrawalLimitBeforeOperate()

timeElapsed = 100

maxWithdrawableLimit_ = (1000 * user_supply) / 10_000 = 600_000; // <- based on comments this is maximum amount
that we can withdraw
expandedWithdrawableAmount_ = (maxWithdrawableLimit_ * timeElapsed) / expandDuration = 300_000; // <- adjusted
withdrawable amount

currentWithdrawalLimit_ = lastWithdrawalLimit_ > expandedWithdrawableAmount_
    ? lastWithdrawalLimit_ - expandedWithdrawableAmount_
    : 0;
currentWithdrawalLimit_ = 5_400_000 - 300_000 = 5_100_000; // lastWithdrawalLimit_ == 5_400_000

minWithdrawalLimit_ = user_supply - maxWithdrawableLimit_ = 6_000_000 - 600_000 = 5_400_000;

currentWithdrawalLimit_ = 5_400_000 // because minWithdrawalLimit_ > currentWithdrawalLimit_

```

Based on this case we can withdraw up to **600_000** tokens, but based on docs and comments we should be able to withdraw only **expandedWithdrawableAmount_**: **300_000**.

Recommendation

It is recommended to resolve inconsistency in docs, comments and calculations or give comments and full flow of **WithdrawalLimit** change if it is intended behaviour.

Description

Functions `fromBigNumber`, `fromBigNumber`, `mulDivNormal`, `decompileBigNumber`, `mulDivBigNumber`, `mulBigNumber`, `divBigNumber` are not used in code.

Recommendation

We recommend considering removing these functions if they will stay unused.

Description

1. Change from `i++` to `unchecked{++i}`:

- [adminModule/main.sol#L40](#)
- [adminModule/main.sol#L60](#)
- [adminModule/main.sol#L303](#)
- [adminModule/main.sol#L335](#)
- [adminModule/main.sol#L406](#)
- [adminModule/main.sol#L430](#)
- [adminModule/main.sol#L621](#)
- [adminModule/main.sol#L846](#)
- [adminModule/main.sol#L864](#)
- [adminModule/main.sol#L897](#)
- [adminModule/main.sol#L918](#)
- [adminModule/main.sol#L970](#)
- [adminModule/main.sol#L988](#)

Recommendation

We recommend considering improving this gas optimizations.

INFORMATIONAL-08	Storage update of exchange prices	Acknowledged
------------------	-----------------------------------	--------------

Description

Based on [Line 439](#) storage update of **exchange prices** occurs only when changes of utilization/supply/borrow ratio hit the threshold because this triggers recalculation of **borrow rate**. But even if **borrow rate** doesn't change, it should be accumulated. In long term linear part of **borrowExchangePrice_increase** would underestimate interests. Ideally, **exchange prices** should be updated every second, but in fact minimal period is one block - 12 seconds.

Recommendation

It is recommended to update **exchange prices** and **last update timestamp** with every call of **operate()** function.

Client's comments

We expect the frequency of updates to be often enough so that the difference in the compounding effect is acceptable. As you point out, on-chain the compounding is always an imperfection. We can adjust the update storage threshold + we can run a bot that triggers **updateExchangePrices()** if e.g. after xyz of time no storage update has happened. The difference for 1 year at 10% rate would be:

- updated every block (12 seconds): $(1+0.1/(365*24*60*6))^{(365*24*60*6)} = 1.105170916043204$
- updated once a day: $(1+0.1/(365))^{(365)} = 1.1051557816162325$
- updated 6 times a day: $(1+0.1/(365*6))^{(365*6)} = 1.1051683949338504$

The difference between updating every block and 6 times a day is 0.002391537616559%

INFORMATIONAL-09	Double calculations	Acknowledged
------------------	---------------------	--------------

Description

At [Lines 474 - 477](#) there is check for skipping transfers, but these calculations are already done at [Lines 583 - 586](#) and **operateAmountIn_** is passed to internal function.

Recommendation

It is recommended to save the value of **operateAmountIn_** in internal **_operate()** function to use it in further condition and to avoid double calculations.

Client's comments

This is true but the **_operate()** method is at the limit of the stack and adding any variable that would store the result causes a Stack too deep error. We would have to introduce a struct, which ends up making things more expensive. So we had to go with double calculating for the special case rather than making the cost overall bigger for all causes. Please let us know if you have an idea about how this could be done without a struct. The problem is this is also a value that must be stored from the beginning of the method to almost the end, so a temporary var would also be blocked for most of the method...

Description

It is possible to set $k1 == k2$ which bypasses the guard condition. In this case, every call of calcRateV2, where **utilization < kink2**, will result in a revert due to a zero division error.

```

if (utilization_ < kink1_) {
  ...
  x1_ = 0; // 0%
  x2_ = kink1_;
} else if (utilization_ < kink2_) {
  ...
  x1_ = kink1_;
  x2_ = kink2_;
} else {
  ...
  x1_ = kink2_;
  x2_ = FOUR_DECIMALS;
}
uint256 slope_ = ((y2_ - y1_) * TWELVE_DECIMALS) / (x2_ - x1_); // zero division

```

Also, revert will happen if **kink1 = 0** or **kink2 = FOUR_DECIMALS**.

Same issue applies for the calcRateV1.

Recommendation

We recommend changing guard condition to

```

if (
  rataDataV2Params_.kink1 >= rataDataV2Params_.kink2 ||
  ...
)

```

and adding sanity checks for the boundary values of the kink variables.

Description

In the comment there is probably a mistake. The comment should be: **outpaces withdrawals**.

In the comment there is probably a mistake. The formula in code: $supply_increase = borrow_increase * (1 - fee)$.

Recommendation

We recommend fixing these typos.

INFORMATIONAL-12	Possible DOS with rate manipulation	Fixed at 4e3423
------------------	--	---------------------------------

Description

In function `calcBorrowRateFromUtilization()` there is a check to limit calculated **rate**, which reverts execution in case of exceeding the value **X16 = 65535**. Utilization can be manipulated and can be more than **100%**. E.g. if protocol has high borrow limit, it can borrow, then transfer tokens on **liquidity** contract and borrow again and repeat this cycle to manipulate the utilization. So there is a possibility for **rate** to exceed **65535** value and in that case main functionality of **liquidity** module will be blocked. **Governance** should update **rate** data to mitigate such effect.

Recommendation

It is recommended to use maximum allowed value - **65535** to not exceed 16 bit and emit event indicating needed changes in rates model instead of reverting execution in case of exceeding **rate** limit.

INFORMATIONAL-13	Using same slot for proxy admin and governance	Acknowledged
------------------	--	--------------

Description

In **infiniteProxy** and in **adminModule** implementation same slot is used for **proxy** admin and **governance**. It is better to separate this roles because they have logically different functions.

Recommendation

It is recommended to have two different slots for **proxy** admin address and **governance** address.

Client's comments

Acknowledged but keeping things simple as both roles are expected to be held by the same Governance for the foreseeable future. If it changes, we can update Liquidity logic accordingly.

INFORMATIONAL-14	Zero address check for _revenueCollector	Fixed at 567875
------------------	---	---------------------------------

Description

In **adminModule** there is no initialization function, so at the beginning **_revenueCollector** will be zero address. Function `collectRevenue()` is open and can be called by anyone, so if **_revenueCollector** is not set, protocol profit could be lost.

Recommendation

It is recommended to add check for zero address of **_revenueCollector** in `collectRevenue()` function.

```

...

if (revenueAmount_ > 0 && _revenueCollector != address(0)) {
    ...
}

...

```


INFORMATIONAL-18	Additional checks for Liquidity's token configs	Acknowledged
<p>Description</p> <p>At <u>Lines 167 - 170</u> asset is checked if it is configured in Liquidity. But this check is insufficient because once token is added to Liquidity its config will be always not null.</p> <p>Recommendation</p> <p>It is recommended to add further check if deployed address of iToken is already added in Liquidity in _userSupplyData mapping and add possibility in Liquidity to stop or to nullify value in _exchangePricesAndConfig.</p> <p>Client's comments</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>Adding the possibility to nullify _exchangePricesAndConfig is very powerful and would only be an option if totalAmounts for that token are 0. We do not expect this case to happen, and if it eventually happens we could upgrade Liquidity to accommodate this need. More importantly, we don't really see an issue here. This check was thought of when createToken() was an open public method to avoid people creating iTokens for unlisted tokens (at Liquidity) but createToken() is auth-protected now, and the flow for listing new tokens will always be (as per docs.md):</p> <ul style="list-style-type: none"> -process to list a token at Liquidity is: 1. Set rate config for token 2. Set token config 3. allow any user. If done in any other way, an error is thrown. -process to create a new Lending iToken is: 1. set up config for underlying asset at Liquidity 2. deploy iToken via LendingFactory createToken() 3. configure user supply config at Liquidity. </div>		

INFORMATIONAL-19	Redundant check	Fixed at 7ba5c8
<p>Description</p> <p>At <u>Line 66</u> rewardsRateModel_ is checked for zero address, but previously there is a check for _rewardsActive. If _rewardsActive is True, then rewardsRateModel_ will be always set.</p> <p>Recommendation</p> <p>It is recommended to remove redundant check.</p>		

INFORMATIONAL-20	Sanity check for uint to int conversion	Fixed at 157ab0
<p>Description</p> <p>In function _depositToLiquidity() and _withdrawFromLiquidity() variable uint256 assets_ is converted to int256. When value of assets_ is bigger than $2^{256} - 1$, this will cause overflow during conversion, so int256 variable will be negative. This may cause unexpected actions on Liquidity layer. Deposit in Lending could actually trigger withdraw from Lending and vice versa during withdraw on Lending.</p> <p>Recommendation</p> <p>It is recommended to add sanity check if converted int256 value is positive or negative, during deposits or withdrawals.</p>		

INFORMATIONAL-21	Code duplicate	Fixed at 1d21cc
<p>Description</p> <p>In function <code>getLiquidityBalance()</code> <code>supplyExchangePrice_</code> is calculated, but there is already a special function to get it from <code>Liquidity</code> - <code>getLiquidityExchangePrice()</code>.</p> <p>Recommendation</p> <p>It is recommended to use <code>getLiquidityExchangePrice()</code> function to get <code>supplyExchangePrice_</code> to avoid code duplication.</p>		

INFORMATIONAL-22	Wrong slot numbers in comments	Fixed at 346fe6
<p>Description</p> <p>In <code>iToken Variables</code> contract variables' slot numbers are indicated in comments. In <code>ERC20Permit</code> section there is only one variable - <code>_nonces</code>. But in OZ library in version 4.8.2 <code>ERC20Permit</code> contract from <code>draft-ERC20Permit.sol</code> has two variables - mapping <code>_nonces</code> and <code>bytes32 private _PERMIT_TYPEHASH_DEPRECATED_SLOT</code>. So slot 6 is occupied by that variable and further variables have wrong slot numbers in comments.</p> <p>Recommendation</p> <p>It is recommended to adjust right slot numbers in comments based on all dependencies and parent contracts.</p>		

INFORMATIONAL-23	Unused imports	Fixed at 00024c
<p>Description</p> <p>The imported files are not used:</p> <ol style="list-style-type: none"> 1. <code>IERC20Permit</code> & <code>BigMath</code> in <code>lending/iToken/main.sol</code>; 2. <code>Structs</code> in <code>lending/lendingFactory.sol</code>; 3. <code>ErrorTypes</code> in <code>vaultT1/vault/core/helpers.sol</code> <p>Recommendation</p> <p>We recommend removing unused imports.</p>		

INFORMATIONAL-24	Redundant input variable	Fixed at b569fb
<p>Description</p> <p>In the methods designed to use underlying tokens, the input variable <code>uint256 assets_</code> is used to pass the amount of tokens. The input is redundant as <code>msg.value</code> is <code>uint256</code> and can be used by itself.</p> <ol style="list-style-type: none"> 1. If removed from the <code>_executeDepositETH</code>, the <code>msg.value < assets_</code> check must be moved to the <code>mintETH</code> function to execute check on <code>previewMint</code> result. That way the <code>depositETH</code> function's input variable <code>uint256 assets_</code> can be lifted. 2. In <code>fundRewardsETH</code> the input variable can be removed. <p>Recommendation</p> <p>We recommend removing redundant variables.</p>		

INFORMATIONAL-25	Redundant type conversion	Fixed at f40cfc
<p>Description</p> <p>The <code>CREATE3.deploy()</code> returns <code>address</code>, but converted to <code>address in createToken</code> invocation.</p> <p>Recommendation</p> <p>We recommend removing redundant type conversion.</p>		
INFORMATIONAL-26	Memory to calldata gas optimization	Fixed at d18590
<p>Description</p> <p>The calldata location is cheaper to use than memory.</p> <p>1. In <code>vaultT1(address supplyToken_, address borrowToken_)</code> the inputs are saved into the memory structure, which is used in the ternary condition.</p> <pre data-bbox="178 994 1921 1216"> constants_.supplyToken = supplyToken_; constants_.supplyDecimals = constants_.supplyToken != NATIVE_TOKEN ? IERC20(supplyToken_).decimals() : 18; constants_.borrowToken = borrowToken_; constants_.borrowDecimals = constants_.borrowToken != NATIVE_TOKEN ? IERC20(borrowToken_).decimals() : 18; </pre> <p>Recommendation</p> <p>We recommend using calldata whenever possible to preserve gas.</p>		
INFORMATIONAL-27	Redundant variable in scope	Fixed at 0f4048
<p>Description</p> <p>In the <code>_addDebtToTickWrite</code>, the <code>tickExistingRawDebt_</code> is defined above the <code>if</code> scope but used only in one branch. It will consume gas to initialize and read data even if it's unnecessary.</p> <p>Recommendation</p> <p>We recommend moving <code>tickExistingRawDebt_</code> to its appropriate scope.</p>		
INFORMATIONAL-28	Code duplication	Acknowledged
<p>Description</p> <p>The <code>mapId</code> calculation for the <code>TickHasDebt</code> structure can be implemented as an internal function, which will improve code readability and most likely decrease contract code size.</p> <p>Recommendation</p> <p>We recommend introducing an internal <code>mapId</code> calculation function.</p> <p>Client's comments</p> <pre data-bbox="178 2398 1921 2576"> Vault code / Fluid in general puts high importance on any gas cost savings, so because contract code size is still below max limit we prefer having slight code duplication in exchange for even tiny gas savings. This applies also to some other parts of duplicated code in Vault protocol (or Liquidity for that matter) e.g. fetching Oracle price etc. </pre>		

INFORMATIONAL-29	Incorrect BigMath sizes in comments	Fixed at 449eba
<p>Description</p> <p>In the description of the variables, a handful of comments say debt factor (27 bits precision & 13 bits expansion), which contradicts its size, the other comment Debt factor = 50 bits (35 bits coefficient 15 bits expansion), and BigMathVault library constant values (COEFFICIENT_SIZE_DEBT_FACTOR = 35 & EXPONENT_SIZE_DEBT_FACTOR = 15).</p> <p>Recommendation</p> <p>We recommend resolving contradictions.</p>		

INFORMATIONAL-30	Double calculation of vault_ address	Acknowledged
<p>Description</p> <p>In function <code>vaultT1()</code>, address vault_ is calculated to be further used in <code>_calculateLiquidityVaultSlots()</code>. But this address was already calculated in function <code>deployVault()</code> at Line 246</p> <p>Recommendation</p> <p>It is recommended to avoid double calculations.</p> <p>Client's comments</p> <p>The calculated vault address would have to be passed in from off-chain and then in deployment scripts. Even more could be passed in from off-chain actually. But the cost here should not be too much as VaultFactory will already be a warm address plus more importantly gas optimization for vault deployment does not matter as much.</p>		

INFORMATIONAL-31	Unnecessary cast to payable address	Fixed at 5265fc
<p>Description</p> <p>At Line 232 <code>address(this)</code> is casted to payable to get contract's balance and to transfer it to LIQUIDITY.</p> <p>Recommendation</p> <p>It is recommended to remove cast to payable, because it is redundant to get contract's balance.</p>		

INFORMATIONAL-32	Unrestricted maxLTV in StETHQueue	Fixed at f5f69e
<p>Description</p> <p>The StETHQueue contract does not implement an upper limit on the maxLTV ratio, other than preventing it from being zero.</p> <pre data-bbox="178 460 1081 845"> function setMaxLTV(uint16 maxLTV_) external onlyAuths { if (maxLTV_ == 0) { // <-- only check for zero revert StETHQueueError(ErrorTypes.StETH__MaxLTVZero); } maxLTV = maxLTV_; emit LogSetMaxLTV(maxLTV_); } </pre> <p>If maxLTV is set to a value higher than HUNDRED_PERCENT, it allows users to borrow an amount of ETH that exceeds the value of their stETH collateral. This scenario opens up an arbitrage opportunity, potentially enabling users to drain ETH up to borrow limit from the Liquidity contract.</p> <p>Recommendation</p> <p>It is recommended to implement a cap on the maxLTV value to ensure it does not exceed HUNDRED_PERCENT.</p>		

INFORMATIONAL-33	Optimization of rate calculation	Fixed at a739e4
<p>Description</p> <p>In function <code>_getChainlinkExchangeRate()</code>, final rate is calculated based on three hops. Conditions, like one at Lines 132 - 134, check if next hop exists. However further calculations are unnecessary if the rate is zero after one of these steps.</p> <p>Recommendation</p> <p>It is recommended to consider adding a check for zero rate and not to continue calculations for the next hops.</p>		

INFORMATIONAL-34	Strict sanity check of TWAP periods	Fixed at 4ad772
<p>Description</p> <p>Condition at Lines 81 - 88 checks for ascending order of seconds ago values. But comparisons are not strict and they allow to have the same values. This may lead to incorrect calculation of TWAP intervals and division by zero.</p> <p>Recommendation</p> <p>It is recommended to make more strict comparisons and not to allow the same value of seconds ago.</p>		

Description

At [Lines 160 - 167](#) `exchangeRate_` is calculated based on mean tick value:

...

```
int24((tickCumulatives[_TWAP_DELTAS_LENGTH + 1] - tickCumulatives[_TWAP_DELTAS_LENGTH]) /
      _UNI_TWAP4_INTERVAL)
```

...

This value can be negative. In that case it should be further processed due to rounding errors as in Uniswap V3

OracleLibrary.sol.

Same issue is in function `_checkTWAPDelta()` at [Line 207](#)

Recommendation

It is recommended to process negative value of mean tick:

...

```
int56 tickCumulativesDelta = tickCumulatives[_TWAP_DELTAS_LENGTH + 1] -
tickCumulatives[_TWAP_DELTAS_LENGTH];
```

```
int24 arithmeticMeanTick = int24(tickCumulativesDelta / _UNI_TWAP4_INTERVAL);
```

```
if (tickCumulativesDelta < 0 && (tickCumulativesDelta % _UNI_TWAP4_INTERVAL != 0)) arithmeticMeanTick--;
```

...

Description

The `uniV3OracleImpl.sol` constructor has `sanity check` that deltas are less than 100% and decreasing. The following comment is attached to this: **all following deltas must be <= than the previous one**. But in the if-else construction, inequalities are not strict.

This leads to incorrect if-else logic with equal deltas and the constructor will revert.

Recommendation

We recommend changing the type of inequalities to strict ones.

INFORMATIONAL-37	Incorrect logic of <code>to_</code> in operate method of Vault	Fixed at 3c57b0
<p>Description</p> <p>In the <code>operate</code> method a user can provide <code>to_ = 0</code> here and as stated it will be changed to <code>msg.sender</code>. However <code>to_</code> is provided as is to the Liquidity Layer(link). In case of <code>to_ = 0</code> it leads to revert.</p> <p>Recommendation</p> <p>We recommend setting <code>to_ = msg.sender</code> if <code>to_</code> equals to <code>0</code>.</p>		

INFORMATIONAL-38	User can bypass the check in flashLoanMultiple	Fixed at 661f9a
<p>Description</p> <p>A user can call <code>flashLoanMultiple</code> with <code>tokensWithAmounts_</code> containing the same token twice, for example, user can bypass the check with <code>[NATIVE_TOKEN, maxFlashLoan()], [NATIVE_TOKEN, maxFlashLoan()]</code>, it will cause the revert in the liquidity layer.</p> <p>Recommendation</p> <p>We recommend making correct revert in this case.</p>		

INFORMATIONAL-39	<code>maxFlashLoan</code> calculates an incorrect amount due to lack of subtracting of <code>userBorrow_</code> from borrow limit.	Fixed at 11047a
<p>Description</p> <p>As <code>maxFlashLoan</code> returns total amount protocol borrow can reach (not borrowable amount in current operation), it does not change the limit through <code>tokensWithAmounts_</code> iteration in <code>flashLoanMultiple</code>.</p> <p>Recommendation</p> <p>We recommend changing <code>maxFlashLoan</code> function.</p> <pre>function maxFlashLoan(address token_) public view returns (uint256) { uint256 userBorrowData_ = _readUserBorrowData(token_); uint256 userBorrow_ = (userBorrowData_ >> LiquiditySlotsLink.BITS_USER_BORROW_AMOUNT) & LiquidityCalcs.X64; - return LiquidityCalcs.calcBorrowLimitBeforeOperate(userBorrowData_, userBorrow_); + return LiquidityCalcs.calcBorrowLimitBeforeOperate(userBorrowData_, userBorrow_) - userBorrow_; }</pre>		

INFORMATIONAL-40	Unused library function	Fixed at 3ee823
<p>Description</p> <p>Function <code>toBigNumber</code> is not used in any part of protocol contracts.</p> <p>Recommendation</p> <p>We recommend removing this function from the library code.</p>		

**STATE
MIND**