Cod3x lend Security Review

Auditors

Saw-mon and Natalie, Lead Security Researcher

Cergyk, Security Researcher

Jonatas Martins, Associate Security Researcher

Report prepared by: Lucas Goiriz

February 20, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Cod3x utilizes a network of lightweight AI agents to interpret user commands, build transactions, and manage queries with the aim of turning transactions that would normally take five applications, two bridges, and 15 minutes of stress into a single swipe to confirm.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Cod3x lend according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 33 days in total, Cod3x engaged with Spearbit to review the cod3x-lend protocol. In this period of time a total of **67** issues were found.

Summary

Project Name	Cod3x
Repository	cod3x-lend
Commit	7dec5d46
Type of Project	DeFi, Lending
Audit Timeline	Dec 3rd to Jan 5th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	8	7	0
Medium Risk	15	13	2
Low Risk	16	13	2
Gas Optimizations	6	5	1
Informational	22	15	7
Total	**67**	**53**	**12**

The Spearbit team reviewed Cod3x's `cod3x-lend` holistically on commit hash da83f1a2 and concluded that all issues were resolved and no new vulnerabilities were introduced.

5 Findings

5.1 High Risk

5.1.1 `minLiquidityRate` formula

Severity: High Risk

Context: MiniPoolDefaultReserveInterestRate.sol#L227-L254, MiniPoolPiReserveInterestRateStrategy.sol#L197-L224

Description/Recommendation: When the M_{proxy} does not have enough liquidity during a mini pool borrow, the following happens:

1. The mini pool P_M borrows an unbacked amount from the reserve in the lending pool P_L.
2. P_M deposits backs into P_L to get a wrapped token and take advantage of interest accrual.
3. Deposit that wrapped token into the corresponding reserve in P_M to get a double-wrapped token.
4. Borrow to the user.

During this process the following amount of debt will be minted for the user in the $M_{proxy}(id_{T_{B,U}})$:

$$\Delta a \oslash_{ray} i_{B,T_{A,U}^{wrapper}}^{t_0,MP}$$

and the following amount of lending share will be minted in $M_{proxy}(id_{T_{A,U}})$ for P_M:

$$\Delta a \oslash_{ray} i_{L,T_{A,U}^{wrapper}}^{t_0,MP}$$

and the following debt position will be minted for P_M in the reserve $(U, true)$ of the P_L:

$$(\Delta a \otimes_{ray} i_{L,U}^{t_0,LP}) \oslash_{ray} i_{B,U}^{t_0,LP}$$

where t_0 is the time the `borrow` was called on the mini pool. Now assume from that time to time t_n:

- The set of interactions with the lending pool is J.
- The set of interactions with the mini pool is K.

Then at time t_n the mini pool P_M needs to payback the lending pool the following amount:

$$A_0 = (\Delta a \otimes_{ray} i_{L,U}^{t_0,LP}) \bigotimes_{ray,J} (1_{ray} + r_{B,U}^{t_j,LP})^{\Delta t_j}$$

and the amount owed by the user when calculated in the main underlying asset with all the mini pool borrow interest and lending pool deposit interest becomes:

$$A_1 = ((\Delta a \bigotimes_{ray,K} (1_{ray} + r_{B,T_{A,U}^{wrapper}}^{t_k,MP})^{\Delta t_k}) \otimes_{ray} i_{L,U}^{t_0,LP}) \bigotimes_{ray,J} (1_{ray} + \Delta t_j r_{L,U}^{t_j,LP})$$

note that $\otimes_{ray}$ is not associative, and swapping the order of components can introduce tiny errors, so we have:

$$A_1 \approx (\Delta a \otimes_{ray} i_{L,U}^{t_0,LP}) \bigotimes_{ray,J} (1_{ray} + \Delta t_j r_{L,U}^{t_j,LP}) \bigotimes_{ray,K} (1_{ray} + r_{B,T_{A,U}^{wrapper}}^{t_k,MP})^{\Delta t_k}$$

The amount of the main underlying assets owed to the mini pool from its share in $M_{proxy}(id_{T_{A,U}})$ at time t_n would be:

$$A_2 = (\Delta a \bigotimes_{ray,K} (1_{ray} + \Delta t_k r_{L,T_{A,U}}^{t_k,MP})) \otimes_{ray} i_{L,U}^{t_0,LP} \bigotimes_{ray,J} (1_{ray} + \Delta t_j r_{L,U}^{t_j,LP})$$

which is roughly:

$$A_2 \approx (\Delta a \otimes_{ray} i_{L,U}^{t_0,LP}) \bigotimes_{ray,J} (1_{ray} + \Delta t_j r_{L,U}^{t_j,LP}) \bigotimes_{ray,K} (1_{ray} + \Delta t_k r_{L,T_{A,U}}^{t_k,MP})$$

Note that all A_0, A_1, A_2 have the common $\Delta a \otimes_{ray} i_{L,U}^{t_0,LP}$ factor (roughly). One might think that A_0 should be able to be repaid by a mix of A_1 and A_2, but we might not be able to always assume that the user would repay its debt on time or at all so we might just want to define the following constraints:

$$A_0 \leq A_2$$

or going a little further:

$$A_0 \leq A_2 \leq A_1$$

The first inequality would give us:

$$\bigotimes_{ray,J} \frac{(1_{ray} + r_{B,U}^{t_j,LP})^{\Delta t_j}}{(1_{ray} + \Delta t_j r_{L,U}^{t_j,LP})} \leq \bigotimes_{ray,K} (1_{ray} + \Delta t_k r_{L,T_{A,U}}^{t_k,MP})$$

This is the more general inequality (and the division is a `ray` division) than the one enforced in the codebase. In the codebase J and K coincide and have only one element and $\Delta t_j = \Delta t_k$ is the 5 day delta time margin normalised and the rates are also normalised with respect to `SECONDS_PER_YEAR`.

And so if there are lots of interactions with the lending pool corresponding to the reserve in the question compared to the one in the mini pool, the mini pool's minted lending shares in itself might not be able to repay the borrowed loans in the lending pool.

For simplicity sake in the equations below assume we are analysing the case where $J = K$ and we only have one $t = \Delta t_j$ and $r_L = r_{L,U}^{t_j,LP}$, $r_B = r_{B,U}^{t_j,LP}$ then define:

$$f(t) = g(t, r_L, r_B) = \frac{(1 + r_B)^t - (1 + r_L t)}{t + r_L t^2} \approx \frac{(r_B - r_L) + (\frac{t-1}{2}) r_B^2 + (\frac{t-1}{2})(\frac{t-2}{3}) r_B^3}{1 + r_L t}$$

$$f(0) = \log(1 + r_B) - r_L \approx (r_B - r_B^2/2 + r_B^3/3) - r_L \approx r_B - r_L$$

$$f(1) \approx \frac{r_B - r_L}{1 + r_L}$$

A few notes.

i. The expressions above are normalised with respect to the `Ray` precision. ii. Binomial approximation is used up to 4 terms according to the codebase. iii. At $t = 0$ the inequality holds since both sides would be 1_{ray} so we don't have to analyse $f(0)$ that much.

One can show that on the interval $[1, \infty)$ the function $f(t)$ is increasing (please double check). A desired property to prove is that $f(t)$ is an increasing function in the positive domain or at least in the domain from `0` up to the `DELTA_TIME_MARGIN` t_m this would give us a guarantee that $f(t_m)$ is the maximum value in the domain $[0, t_m]$. So based above one needs to take the maximum of $f(0)$ and $f(t_m)$ to find the maximum in the set $[0, t_m] \cap \mathbb{Z}$ let's call this value f_{max}. If we prove or guarantee the following two invariants we can prove that $A_0 \leq A_2$:

1. $J \subset K$, this still needs to be enforced by making sure whenever the interest rates in the reserve $(U, true)$ are updated in the lending pool the corresponding tranched reserves' interest rates are updated in all the mini pools using that $T^{wrapper}_{A,U}$. Although this might be costly operation.
2. $f_{max} \leq r^{t_k,MP}_{L,T_{A,U}}$, this has been partially check in the code in this context although `minLiquidityRate` is not compared against $f(0)$.
3. The interactions with the lending pool are not more than `DELTA_TIME_MARGIN` far apart. We need to make sure for all j, $\Delta t_j \leq t_m$.

Footnote: The second inequality $A_2 \leq A_1$ gives us:

$$\bigotimes_{ray,K}(1_{ray} + \Delta t_k r^{t_k,MP}_{L,T_{A,U}}) \leq \bigotimes_{ray,K}(1_{ray} + r^{t_k,MP}_{B,T^{wrapper}_{A,U}})^{\Delta t_k}$$

where roughly assuming $r^{t_k,MP}_{L,T_{A,U}} \leq r^{t_k,MP}_{B,T^{wrapper}_{A,U}}$ should prove the above. The `IRM` implementations already follow this invariant when picking `currentLiquidityRate` and `currentVariableBorrowRate`.

Cod3x: Addressed in PR 31.

5.1.2 `RewardsController` uses inconsistent scaling in `handleAction` and can lead to transfer DoS to/from minipool market

Severity: High Risk

Context: RewardsController.sol#L139

Description: `RewardController.handleAction` has a special case to handle the case when user is a minipool market:

- RewardsController.sol#L134-L145:

```
if (_isAtokenERC6909[user] == true) {
    (uint256 assetID,
    ) =
        IAERC6909(user).getIdForUnderlying(IAToken(msg.sender).WRAPPER_ADDRESS());
    // For trancheATokens we calculate the total supply of the AERC6909 ID for the assetID.
    // We subtract the current balance.
    uint256 totalSupplyAsset = IAERC6909(user).scaledTotalSupply(assetID); // <<<
    uint256 diff = totalSupplyAsset - userBalance;
    _totalDiff[msg.sender] =
        _totalDiff[msg.sender] - lastReportedDiff[msg.sender][user] + diff;
    lastReportedDiff[msg.sender][user] = diff;
    userBalance = totalSupplyAsset;
}
```

We can see that the function fetches `scaledTotalSupply` in order to compute the difference between the shares of MLP aToken which are due to the minipool and what is actually held by the minipool. These "shares" which are not accounted for in `aToken.scaledTotalSupply` should be rewarded, and thus this is why `_totalDiff[msg.sender]` is computed and then added to `totalSupply` for updating reward state:

- RewardsController.sol#L146-L148:

```
_updateUserRewardsPerAssetInternal(
    msg.sender,
    user,
    userBalance,
    totalSupply + _totalDiff[msg.sender] // <<<
);
```

Unfortunately, it is inaccurate to use `IAERC6909(user).scaledTotalSupply` in this context, since the total supply is scaled according to the minipool market index, while all other amounts are only scaled according to main lending pool index. To showcase how it can lead to DOS of transfers from the ERC6909 market, we will use a simple example:

Scenario:

- Preconditions:

Params	Value
asset	`aWETH`
MLP liquidity index	`1`
minipool liquidity index	`1.05`
initial total supply	`0`

- Steps:
  - Alice mints 100 aWETH by depositing in the main lending pool.
  - Alice deposits the 100 aWETH into minipool, and is minted 95.2 shares of `aWETH6909`.

    After this step, `aWETH6909.scaledTotalSupply` is `95.2e18`.
- Alice attempts to withdraw the 100 aWETH, but the following values are used in `RewardController.handleAction` during transfer:
  - IncentivizedERC20.sol#L191-L199:

```
    if (address(_getIncentivesController()) != address(0)) {
        uint256 currentTotalSupply = _totalSupply;
        _getIncentivesController().handleAction(sender, currentTotalSupply,
        ↪  oldSenderBalance); // <<<
        if (sender != recipient) {
            _getIncentivesController().handleAction(
                recipient, currentTotalSupply, oldRecipientBalance
            );
        }
    }
```

Params	Value
sender	`aWETH6909`
currentTotalSupply	`100e18`
oldSenderBalance	`100e18`

Which means `handleAction` will underflow (`scaledTotalSupply == 95.2e18`):

  - RewardsController.sol#L134-L145:

```
if (_isAtokenERC6909[user] == true) {
    (uint256 assetID,
    ) =
        IAERC6909(user).getIdForUnderlying(IAToken(msg.sender).WRAPPER_ADDRESS());
    // For trancheATokens we calculate the total supply of the AERC6909 ID for the assetID.
    // We subtract the current balance.
    uint256 totalSupplyAsset = IAERC6909(user).scaledTotalSupply(assetID);
    uint256 diff = totalSupplyAsset - userBalance; // <<<
    _totalDiff[msg.sender] =
        _totalDiff[msg.sender] - lastReportedDiff[msg.sender][user] + diff;
    lastReportedDiff[msg.sender][user] = diff;
    userBalance = totalSupplyAsset;
}
```

Alice is unable to withdraw her funds.

Recommendation: We should use `totalSupply` to keep units consistent:

- RewardsController.sol#L134-L145:

```
if (_isAtokenERC6909[user] == true) {
    (uint256 assetID,
    ) =
        IAERC6909(user).getIdForUnderlying(IAToken(msg.sender).WRAPPER_ADDRESS());
    // For trancheATokens we calculate the total supply of the AERC6909 ID for the assetID.
    // We subtract the current balance.
    uint256 totalSupplyAsset = IAERC6909(user).totalSupply(assetID);
    uint256 diff = totalSupplyAsset - userBalance;
    _totalDiff[msg.sender] =
        _totalDiff[msg.sender] - lastReportedDiff[msg.sender][user] + diff;
    lastReportedDiff[msg.sender][user] = diff;
    userBalance = totalSupplyAsset;
}
```

Note that it may seem a bit counterintuitive, because this total supply includes shares of lending pool which are accrued as interest, and have never been "minted" yet.

Cod3x: Fixed in commit 92f8ffaf.

Spearbit: Fix verified.

5.1.3 Index can reach `type(uint104).max` when asset `totalSupply` is dust and DoS aToken transfers indifinetely

Severity: High Risk

Context: RewardsDistributor.sol#L501

Description: The reward formula in Reward Distributors (`RewardsDistributor.sol` and `RewardsDistributor6909.sol`) uses an index to track accrual of rewards per unit of aToken held. The size for the index is set to be `uint104`, and it should be well enough for cases when `assetTotalSupply >= 10**assetDecimals` or decimals is a low value. However in the case totalSupply is dust, even for a short amount of time, the index can reach `type(uint104).max`, and no further reward accrual can happen:

- RewardsDistributor.sol#L279:

```
if (newIndex != oldIndex) {
    require(newIndex <= type(uint104).max, "Index overflow"); // <<<
    //optimization: storing one after another saves one SSTORE
    rewardConfig.index = uint104(newIndex);
    rewardConfig.lastUpdateTimestamp = uint32(block.timestamp);
    emit AssetIndexUpdated(asset, reward, newIndex);
} else {
    rewardConfig.lastUpdateTimestamp = uint32(block.timestamp);
}
```

The formula for computing the index is given below:

- RewardsDistributor.sol#L501:

```
uint256 currentTimestamp =
    block.timestamp > distributionEnd ? distributionEnd : block.timestamp;
uint256 timeDelta = currentTimestamp - lastUpdateTimestamp;
return (emissionPerSecond * timeDelta * (10 ** decimals)) / totalBalance + currentIndex;
```

`emissionPerSecond` is the number of reward tokens to emit globally per second.

`decimals` is the decimals precision for asset token considered.

`totalBalance` is the total supply of the asset token considered.

Scenario:

- Preconditions:

Parameter	Value
Asset token	aWeth (`decimals: 18`)
Reward token	DAI (`decimals: 18`)
Total reward amount (`A`)	1000 DAI
Total asset supply	1
Time elapsed (`t`)	12 sec (~1 block)
Distribution duration (`T`)	1 month

Very reasonable values except for `total asset supply` which may need some stars to align.

- Index calculation: First let's calculate `emissionPerSecond` denoted `r`:

$$r = A/T = (1000 \times 10^{18})/262800 \ = 3.8 \times 10^{14}$$

Which yields the index value:

$$i = r.t.10^{d} = (3.8 \times 10^{14}) \times 12 \times 10^{18} \ = 4.56 \times 10^{34} \gg 2^{104}$$

Recommendation: Multiple recommendations can be considered:

- Increasing index size: When `A = 1_000_000e18` and `d = 18`, index would be safe from reverting using `uint140`.
- Do not accrue index when normalized total supply is below a threshold (which should not happen anyway in any reasonable case):

```
    if (
        emissionPerSecond == 0 ||
        totalBalance == 0 ||
+       (decimals == 18 && totalBalance <= UPDATE_THRESHOLD) || //@audit ok to be below threshold
 ↪  for low decimal tokens
        lastUpdateTimestamp == block.timestamp
        || lastUpdateTimestamp >= distributionEnd
    ) {
        return currentIndex;
    }
```

Additionally to avoid reverting at any costs during action handling, overflow could be allowed so the index would naturally wrap around (it would be safe since we only use index differences when computing rewards).

Cod3x: Fixed in PR 34.

Spearbit: Fix verified.

5.1.4 In `updateAToken` and `updateVariableDebtToken` of the `LendingPoolConfigurator` `encodedCall` is constructed incorrectly

Severity: High Risk

Context: LendingPoolConfigurator.sol#L173-L183, LendingPoolConfigurator.sol#L206-L215

Description: In `updateAToken` and `updateVariableDebtToken` of the `LendingPoolConfigurator` `encodedCall` is constructed incorrectly:

```
bytes memory encodedCall = abi.encodeWithSelector(
    /*...*/selector,
    cachedPool,
    // ...
    input.asset,
    input.incentivesController,
    decimals, // <--- `reserveType` is missing after here
    input.name,
    input.symbol,
    input.params
);
```

and thus `RESERVE_TYPE`, `name`, `symbol` and `params` will be set incorrectly in the proxy contracts. This will cause all the following calls to query or update data for a wrong reserve in the lending pool:

```
pool.function(_underlyingAsset, RESERVE_TYPE, /*...*/)
```

Recommendation: Add the missing `reserveType` parameter and also make sure to instead of `abi.encodeWithSelector` use `abi.encodeCall` to avoid potential future mistakes regarding typos and incorrect parameter types.

Cod3x: Fixed in commit 8bc4648c.

Spearbit: Fix verified.

5.1.5 `ATokenERC6909.totalSupply` for debt tokens uses the liquidity index

Severity: High Risk

Context: ATokenERC6909.sol#L615-L625

Description: The `totalSupply` function is defined as:

```
function totalSupply(uint256 id) public view override returns (uint256) {
    uint256 currentSupplyScaled = super.totalSupply(id);

    if (currentSupplyScaled == 0) {
        return 0;
    }

    return currentSupplyScaled.rayMul(
        POOL.getReserveNormalizedIncome(_underlyingAssetAddresses[id])
    );
}
```

which assumes the `id` provided is always an `aToken` id and thus uses the `POOL.getReserveNormalizedIncome(_underlyingAssetAddresses[id])` as an index.

1. This function needs to be used by other on and off chain agent to query the correct amount for debt tokens as well.
2. It is used in `MiniPoolPiReserveInterestRateStrategy.getCurrentInterestRates()` to calculate the utilisation rate, and so wrong values are returned.
3. It is used in `_afterTokenTransfer` to supply the `oldSupply` to `INCENTIVES_CONTROLLER`. And thus for debt tokens with incentives incorrect values would be provided.

Fortunately enough the correct value of `totalVariableDebt` was calculated manually in `MiniPoolReserveLogic.updateInterestRates` and thus the state transition for both the default and `PiReserve InterestRateStrategy`s use the correct calculation:

```
vars.totalVariableDebt = IAERC6909(reserve.aTokenAddress).scaledTotalSupply(
        (reserve.variableDebtTokenID)
    ).rayMul(reserve.variableBorrowIndex)
```

If the above optimisation would have not been used the issue would have been more severe.

Recommendation: Make sure `totalSupply` checks whether the `id` is an `aToken` or `debtToken`:

```
/**
 * @notice Gets the total supply for a token ID.
 * @param id The token ID.
 * @return The total supply scaled by normalized income/debt.
 */
function totalSupply(uint256 id) public view override returns (uint256) {
    uint256 currentSupplyScaled = super.totalSupply(id);

    if (currentSupplyScaled == 0) {
        return 0;
    }

    uint256 index = 0;

    if (isDebtToken(id)) {
        index = POOL.getReserveNormalizedVariableDebt(_underlyingAssetAddresses[id]);
    } else {
        index = POOL.getReserveNormalizedIncome(_underlyingAssetAddresses[id]);
    }

    return currentSupplyScaled.rayMul(index);
}
```

Cod3x: Fixed in commit f8cd1275.

Spearbit: Fix verified.

5.1.6 Minipools can borrow from lending pool reserves that are not borrowable

Severity: High Risk

Context: BorrowLogic.sol#L276-L283.

Description: In this context unlike the `borrow` flow of the lending pool we don't check whether:

- The reserve is not frozen. This check is later performed when one deposits assets into the lending pool again when calling:

```
ILendingPool(vars.LendingPool).deposit(
underlying, true, vars.amountReceived, address(this)
);
```

- The reserve can be borrowed from.

And also based on the assumption that it is an unbacked borrow the following checks are omitted:

- Checking that there are some collaterals.
- The position of the mini pool is healthy (checking the health factor).
- The amount being borrowed does not exceed what is allowed based on the total collateral and average LTV and the total borrowed amount.

Recommendation: Make sure `executeMiniPoolBorrow` checks whether the reserve can be borrowed from `reserve.configuration.getBorrowingEnabled()` should return `true`.

Cod3x: Fixed in commit de7bbdc6.

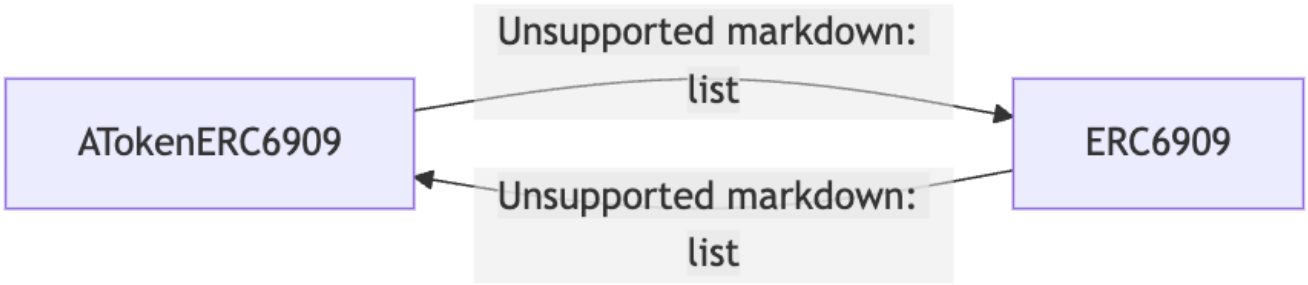
Spearbit: Fix verified.

5.1.7 Mix of scaled and non-scaled parameters in `_afterTokenTransfer`

Severity: High Risk

Context: ATokenERC6909.sol#L396-L432.

Description: Mixing of scaled an unscaled parameters in arithmetic (addition / subtraction) operations has been used in `_afterTokenTransfer`. The `amount` input to `_afterTokenTransfer` comes from `super.{transfer, transferFrom, _mint, _burn}` of `ERC6909` which is a scaled down amount (scaled down by the corresponding `index`, `x.rayDiv(i)`).



But the following parameters are not scaled down and include the `index`s:

```
uint256 oldSupply = totalSupply(id); // not scaled (only at this line later when `_decrementTotalSupply`
↪  or `_incrementTotalSupply` is called it would be scaled
uint256 oldFromBalance = balanceOf(from, id); // not scaled
uint256 oldToBalance = balanceOf(to, id); // not scaled
// ...
```

Here is the summary line by line:

```
function _afterTokenTransfer(
    address from,
    address to,
    uint256 id,
    uint256 amount // scaled
)
    internal
    override
{
    uint256 oldSupply = totalSupply(id); // not scaled
    uint256 oldFromBalance = balanceOf(from, id); // not scaled
    uint256 oldToBalance = balanceOf(to, id); // not scaled
    if (from == address(0) && to != address(0)) {
        oldSupply = _incrementTotalSupply(id, amount); // scaled
        oldToBalance = oldToBalance - amount; // mix: (not scaled) - (scaled)
        oldFromBalance = 0;
        if (address(INCENTIVES_CONTROLLER) != address(0)) {
            INCENTIVES_CONTROLLER.handleAction(id, to,
                oldSupply, // scaled
                oldToBalance // mix: (not scaled) - (scaled)
            );
        }
    } else if (to == address(0) && from != address(0)) {
        oldSupply = _decrementTotalSupply(id, amount); // scaled
        oldFromBalance = oldFromBalance + amount; // mix: (not scaled) + (scaled)
        oldToBalance = 0;
        if (address(INCENTIVES_CONTROLLER) != address(0)) {
            INCENTIVES_CONTROLLER.handleAction(id, from,
                oldSupply, // scaled
                oldFromBalance // mix: (not scaled) + (scaled)
            );
        }
    }
    else {
        oldFromBalance = oldFromBalance + amount; // mix: (not scaled) + (scaled)
        oldToBalance = oldToBalance - amount; // mix: (not scaled) - (scaled)
        if (address(INCENTIVES_CONTROLLER) != address(0)) {
            INCENTIVES_CONTROLLER.handleAction(id, from,
                oldSupply, // not scaled
                oldFromBalance // mix: (not scaled) + (scaled)
            );

            if (from != to) {
                INCENTIVES_CONTROLLER.handleAction(id, to,
                    oldSupply, // not scaled
                    oldToBalance // mix: (not scaled) - (scaled)
                );
            }
        }
    }
}
```

For the `AToken`s in `IncentivizedERC20`, the `_transfer`, `_mint` and `_burn` functions supply the *scaled* total supply and balances to the incentives controller, which means the units of the non-rebasing aToken are used there.

Recommendation: Make sure scaled units/amounts are supplied to `INCENTIVES_CONTROLLER.handleAction`:

```
diff --git a/contracts/protocol/tokenization/ERC6909/ATokenERC6909.sol
↪  b/contracts/protocol/tokenization/ERC6909/ATokenERC6909.sol
index 2de411d..fc521b0 100644
--- a/contracts/protocol/tokenization/ERC6909/ATokenERC6909.sol
```

```
+++ b/contracts/protocol/tokenization/ERC6909/ATokenERC6909.sol
@@ -390,21 +390,21 @@ contract ATokenERC6909 is IncentivizedERC6909, VersionedInitializable {
      * @param from The address tokens are transferred from.
      * @param to The address tokens are transferred to.
      * @param id The token ID being transferred.
-     * @param amount The amount being transferred.
+     * @param amount The amount being transferred in shares.
      * @dev Updates incentives based on transfer type (mint/burn/transfer).
+     * @dev this hook gets called from solday's `ERC6909` which only deals with shares
      */
     function _afterTokenTransfer(address from, address to, uint256 id, uint256 amount)
         internal
         override
     {
-        uint256 oldSupply = totalSupply(id);
-        uint256 oldFromBalance = balanceOf(from, id);
-        uint256 oldToBalance = balanceOf(to, id);
+        uint256 oldSupply = super.totalSupply(id);
+        uint256 oldFromBalance = super.balanceOf(from, id);
+        uint256 oldToBalance = super.balanceOf(to, id);
         //If the token was minted.
         if (from == address(0) && to != address(0)) {
             oldSupply = _incrementTotalSupply(id, amount);
             oldToBalance = oldToBalance - amount;
-            oldFromBalance = 0;
             if (address(INCENTIVES_CONTROLLER) != address(0)) {
                 INCENTIVES_CONTROLLER.handleAction(id, to, oldSupply, oldToBalance);
             }
@@ -412,7 +412,6 @@ contract ATokenERC6909 is IncentivizedERC6909, VersionedInitializable {
         } else if (to == address(0) && from != address(0)) {
             oldSupply = _decrementTotalSupply(id, amount);
             oldFromBalance = oldFromBalance + amount;
-            oldToBalance = 0;
             if (address(INCENTIVES_CONTROLLER) != address(0)) {
                 INCENTIVES_CONTROLLER.handleAction(id, from, oldSupply, oldFromBalance);
             }
```

Moreover this hook can be moved to `IncentivizedERC6909` to mimic the pattern from the `IncentivizedERC20` counterpart.

In general it might make sense to use Solidity's type system and override arithmetic operations to guarantee type safety during compilation and avoid mistakes like above.

Cod3x: Fixed in commit 92f8ffaf.

Spearbit: Fix verified.

5.1.8 DoSing vault `rehypothecation` through flashloans

Severity: High Risk

Context: AToken.sol#L388

Description: During a flash loan, the `AToken` contract transfers the underlying amount to the user through the `transferUnderlyingTo` function, which withdraws tokens from the vault when using it:

```
function transferUnderlyingTo(address target, uint256 amount)
  external
  override
  onlyLendingPool
  returns (uint256)
{
  _rebalance(amount);
  _underlyingAmount = _underlyingAmount - amount;
  IERC20(_underlyingAsset).safeTransfer(target, amount);
  return amount;
}
```

After executing the receiver logic, the underlying tokens are transferred back to the `AToken` contract, but the rebalance is not triggered. This could lead to a situation where the vault is left without tokens, causing a DOS in vault rehypothecation:

```
function handleRepayment(address user, address onBehalfOf, uint256 amount)
  external
  override
  onlyLendingPool
{
  _underlyingAmount = _underlyingAmount + amount;
}
```

A full fuzzing test was provided by Cod3x team and can be found in the echidna branch.

Recommendation: Consider calling the `_rebalance` function exclusively during flashloan operations, since triggering `_rebalance` during liquidation or repayment flows may cause another DOS if the vault is paused or has reached its deposit cap.

Cod3x: Fixed in commit 4605e906.

Spearbit: Fix verified.

5.2 Medium Risk

5.2.1 Rounding directions

Severity: Medium Risk

Context: GenericLogic.sol#L236-L242, MiniPoolGenericLogic.sol#L202-L209

Description/Recommendation:

- GenericLogic.sol#L236-L242, MiniPoolGenericLogic.sol#L202-L209: Make sure all operations round-up when calculating `totalDebtInETH`.
- AToken.sol#L184-L185: Best to round-up `amountScaled`.
- ValidationLogic.sol#L205-L212, MiniPoolValidationLogic.sol#L208-L214: It would be best to round up the `vars.amountOfCollateralNeededETH` or instead check:

```
vars.userBorrowBalanceETH + validateParams.amountInETH <=
↪  (vars.userCollateralBalanceETH).percentMul(vars.currentLtv)
```

Cod3x: Fixed in commit 1ee6d04b.

Spearbit: All the above except the AToken.sol#L184-L185 have been fixed in commit 1ee6d04b.

AToken.sol#L184-L185 has been acknowledged.

5.2.2 `setFlowLimit` does not change the indices and interest rates

Severity: Medium Risk

Context: MiniPoolAddressProvider.sol#L349-L355

Description: Changing the flow limit of a reserve in for a mini pool affects its virtual available liquidity and thus one should updates the interest rates and the indices for that reserve in the mini pool.

Moreover setting the flow limit to a non-zero value or resetting it to `0` can trigger calculation the minimum liquidity rate calculation and this adds another reason as to why calling `setFlowLimit` should trigger updating the interest rates and indices.

Recommendation: Make sure when `setFlowLimit` is called the indices and interest rates are updated.

Cod3x: Fixed in commit 696a5fa4.

Spearbit: Fix verified.

5.2.3 Updating reserve factor should update interest rates

Severity: Medium Risk

Context: LendingPoolConfigurator.sol#L443-L455, MiniPoolConfigurator.sol#L167-L178, MiniPoolConfigurator.sol#L463-L474

Description: The reserve factor is the percentage of borrow interest which is attributed to the protocol (goes to reserve). It directly influences ratio between borrow rate and liquidity rate as can be seen in `getLiquidityRate` implementation:

- BasePiReserveRateStrategy.sol#L291-L299:

```
function getLiquidityRate(
    uint256 currentVariableBorrowRate,
    uint256 utilizationRate,
    uint256 reserveFactor
) internal pure returns (uint256) {
    return currentVariableBorrowRate.rayMul(utilizationRate).percentMul(
        PercentageMath.PERCENTAGE_FACTOR - reserveFactor // <<<
    );
}
```

It can be changed directly by admin by calling `setCod3xReserveFactor`:

- LendingPoolConfigurator.sol#L443-L455:

```
function setCod3xReserveFactor(address asset, bool reserveType, uint256 reserveFactor)
    external
    onlyPoolAdmin
{
    DataTypes.ReserveConfigurationMap memory currentConfig =
        pool.getConfiguration(asset, reserveType);


    currentConfig.setCod3xReserveFactor(reserveFactor);


    pool.setConfiguration(asset, reserveType, currentConfig.data);


    emit ReserveFactorChanged(asset, reserveType, reserveFactor);
}
```

However since calling this function does not update interest rates, this could lead to an insolvency towards lenders in the case where the reserve factor is increased.

Indeed during next accrual, current liquidity rate will be applied to liquidity index, and at the same time the new reserve factor will be taken out of accrued borrow interest:

- ReserveLogic.sol#L284-L304:

```
function _updateIndexes(
    DataTypes.ReserveData storage reserve,
    uint256 scaledVariableDebt,
    uint256 liquidityIndex,
    uint256 variableBorrowIndex,
    uint40 timestamp
) internal returns (uint256, uint256) {
    uint256 currentLiquidityRate = reserve.currentLiquidityRate;


    uint256 newLiquidityIndex = liquidityIndex;
    uint256 newVariableBorrowIndex = variableBorrowIndex;


    // Only cumulating if there is any income being produced.
    if (currentLiquidityRate != 0) {
        uint256 cumulatedLiquidityInterest =
            MathUtils.calculateLinearInterest(currentLiquidityRate, timestamp);
        newLiquidityIndex = cumulatedLiquidityInterest.rayMul(liquidityIndex);
        require(newLiquidityIndex <= type(uint128).max, Errors.RL_LIQUIDITY_INDEX_OVERFLOW);


        reserve.liquidityIndex = uint128(newLiquidityIndex);
    }
    // ...
}
```

- MiniPoolReserveLogic.sol#L251-L269:

```
function _mintToTreasury(
    DataTypes.MiniPoolReserveData storage reserve,
    uint256 scaledVariableDebt,
    uint256 previousVariableBorrowIndex,
    uint256 newLiquidityIndex,
    uint256 newVariableBorrowIndex,
    uint40
) internal {
    MintToTreasuryLocalVars memory vars;

    vars.cod3xReserveFactor = reserve.configuration.getCod3xReserveFactor(); // <<<
    vars.minipoolOwnerReserveFactor = reserve.configuration.getMinipoolOwnerReserveFactor();


    if (vars.cod3xReserveFactor == 0 && vars.minipoolOwnerReserveFactor == 0) {
        return;
    }


    // Calculate the last principal variable debt.
    vars.previousVariableDebt = scaledVariableDebt.rayMul(previousVariableBorrowIndex);

    // ...
}
```

Recommendation: Pool should be touched when updating reserve factors (`Cod3x` and `MinipoolOwner`):

- LendingPoolConfigurator.sol#L443-L455:

```
function setCod3xReserveFactor(address asset, bool reserveType, uint256 reserveFactor)
    external
    onlyPoolAdmin
{
+   //@audit please note that this function needs to be implemented, because there is no endpoint
↪   which can be called with zero amount
+   pool.updateState();
+
    DataTypes.ReserveConfigurationMap memory currentConfig =
        pool.getConfiguration(asset, reserveType);


    currentConfig.setCod3xReserveFactor(reserveFactor);


    pool.setConfiguration(asset, reserveType, currentConfig.data);


    emit ReserveFactorChanged(asset, reserveType, reserveFactor);
}
```

Cod3x: Fixed in commit 4bbb4351.

Spearbit: Fix verified.

5.2.4 WETH9 market cannot be deployed due to flawed `_determineIfAToken` logic

Severity: Medium Risk

Context: ATokenERC6909.sol#L440-L446

Description: When deploying a minipool, underlying tokens are checked to see if they are `aTokens` from the main lending pool.

- ATokenERC6909.sol#L440-L446:

```
function _determineIfAToken(address underlying, address MLP) internal view returns (bool) {
    try IAToken(underlying).getPool() returns (address pool) {
        return pool == MLP;
    } catch {
        return false;
    }
}
```

However this logic is flawed for tokens which have a `fallback` function such as `WETH9`. Indeed the try/catch clause is here to handle the revert in the case where the function `getPool` does not exist on the token; But in the case of `WETH9` the fallback function is executed which attempts a deposit. As a result, an EVM error (attempt to write to storage during `staticcall`) is thrown instead of a revert, and consumes all of the remaining gas in the context.

Note that in `Oracle`, limiting gas passed is used as a workaround for this:

Oracle.sol#L130-L137:

```
// Check if `asset` is an aToken.
try ATokenNonRebasing(asset).UNDERLYING_ASSET_ADDRESS{gas: 4000}() returns (
    address underlying_
) {
    underlying = underlying_;
} catch {
    underlying = asset;
}
```

> And while it works, it is not a very robust solution (for example `aToken` logic is upgraded, and needs more than 4000 gas to get underlying asset).

Impact: When attempting to deploy minipools which have `WETH9` as underlying, the call will always revert due to out of gas.

> Please note that due to EIP-150, only 63/64 of gas is passed on to the `IAToken(underlying).getPool()`, so the call can technically be made to succeed by wasting an enormous amount of gas.

Recommendation: Use a low-level `call` (as opposed to the `staticcall` generated by solidity due to using a view function) to get the pool value. This way in the case of the `WETH9` fallback function, the function would succeed but would not yield a `uint` value, and would be correctly treated as a non-aToken.

Cod3x: Fixed `_determineIfAToken` in commits a0b5ca43 and 208aa500. Fixed oracle in commits 899875bc and c37de4d0.

Spearbit: Fix verified.

5.2.5 Wrong oracle timeout value is used for `Oracle.getAssetPrice`

Severity: Medium Risk

Context: Oracle.sol#L153

Description: `Oracle.getAssetPrice` uses a mapping of oracle addresses and timeouts per asset. These are being initialized alongside each other when calling `_setAssetsSources`:

- Oracle.sol#L99-L110:

```
function _setAssetsSources(
    address[] memory assets,
    address[] memory sources,
    uint256[] memory timeouts
) internal {
    require(assets.length == sources.length, Errors.O_INCONSISTENT_PARAMS_LENGTH);
    for (uint256 i = 0; i < assets.length; i++) {
        _assetsSources[assets[i]] = IChainlinkAggregator(sources[i]); // <<<
        _assetToTimeout[assets[i]] = timeouts[i] == 0 ? type(uint256).max : timeouts[i]; // <<<
        emit AssetSourceUpdated(assets[i], sources[i]);
    }
}
```

However when calling `getAssetPrice` different mapping entries will be used for tranched tokens:

- Oracle.sol#L127-L168:

```
function getAssetPrice(address asset) public view override returns (uint256) {
    address underlying;


    // Check if `asset` is an aToken.
    try ATokenNonRebasing(asset).UNDERLYING_ASSET_ADDRESS{gas: 4000}() returns (
        address underlying_
```

```
    ) {
        underlying = underlying_;
    } catch {
        underlying = asset;
    }


    IChainlinkAggregator source = _assetsSources[underlying]; // <<<
    uint256 finalPrice;


    if (underlying == BASE_CURRENCY) {
        finalPrice = BASE_CURRENCY_UNIT;
    } else if (address(source) == address(0)) {
        finalPrice = _fallbackOracle.getAssetPrice(underlying);
    } else {
        (uint80 roundId, int256 price, uint256 startedAt, uint256 timestamp,) =
            IChainlinkAggregator(source).latestRoundData();


        // Chainlink integrity checks.
        if (
            roundId == 0 ||
            timestamp == 0 ||
            timestamp > block.timestamp ||
            price <= 0 ||
            startedAt == 0 ||
            //@audit _assetToTimeout[asset] is used instead of _assetToTimeout[underlying]
            block.timestamp - timestamp > _assetToTimeout[asset] // <<<
        ) {
            require(address(_fallbackOracle) != address(0), Errors.O_PRICE_FEED_INCONSISTENCY);
            finalPrice = _fallbackOracle.getAssetPrice(underlying);
        } else {
            finalPrice = uint256(price);
        }
    }


    // If `asset` is an aToken then convert the price from asset to share.
    if (asset != underlying) {
        return ATokenNonRebasing(asset).convertToAssets(finalPrice);
    } else {
        return finalPrice;
    }
}
```

Impact: This will lead to invalidation of valid oracle data (in case the timeout is not set for `asset`).

Recommendation: Use underlying entry for `_assetToTimeout`:

```
  // Chainlink integrity checks.
  if (
      roundId == 0 ||
      timestamp == 0 ||
      timestamp > block.timestamp ||
      price <= 0 ||
      startedAt == 0 ||
-     block.timestamp - timestamp > _assetToTimeout[asset]
+     block.timestamp - timestamp > _assetToTimeout[underlying]
  ) {
      require(address(_fallbackOracle) != address(0), Errors.O_PRICE_FEED_INCONSISTENCY);
      finalPrice = _fallbackOracle.getAssetPrice(underlying);
  } else {
      finalPrice = uint256(price);
  }
```

Cod3x: Fixed in commit d73865d3.

Spearbit: Fix verified.

5.2.6 Usage of IERC20 methods would fail on some tokens due to lack of return of boolean value

Severity: Medium Risk

Context: MiniPool.sol#L228

Description: In some contracts, `IERC20` methods (`transfer`/`approve`) are used, and will fail when used with some tokens. Most notable and popular example of this is `USDT` on ethereum mainnet, which lacks a boolean return value for both of these methods. This means that the solidity call to `USDT` will revert when attempting to decode return value.

- MiniPool.sol#L228:

```
  IERC20(underlying).approve(vars.LendingPool, vars.amountReceived);
```

- RewardForwarder.sol#L146:

```
  IERC20(rewardToken).transfer(forwarder, amount);
```

- ATokenERC6909.sol#L238:

```
  IERC20(_underlyingAssetAddresses[id]).transfer(to, amount);
```

Recommendation: Use `safeTransfer` / `forceApprove` from `SafeERC20` library instead:

- MiniPool.sol#L228:

```
  - IERC20(underlying).approve(vars.LendingPool, vars.amountReceived);
  + IERC20(underlying).forceApprove(vars.LendingPool, vars.amountReceived);
```

- RewardForwarder.sol#L146:

```
  - IERC20(rewardToken).transfer(forwarder, amount);
  + IERC20(rewardToken).safeTransfer(forwarder, amount);
```

- ATokenERC6909.sol#L238:

```
  - IERC20(_underlyingAssetAddresses[id]).transfer(to, amount);
  + IERC20(_underlyingAssetAddresses[id]).safeTransfer(to, amount);
```

Cod3x: Fixed in commit d58e3c76.

Spearbit: Fix verified.

5.2.7 Accounting available flow as available liquidity will lead to underestimate utilization

Severity: Medium Risk

Context: MiniPoolPiReserveInterestRateStrategy.sol#L106-L108

Description: When computing available liquidity for minipools, available flow from main lending pool is counted as available liquidity.

- MiniPoolPiReserveInterestRateStrategy.sol#L91-L112:

```
function getAvailableLiquidity(address asset, address aToken)
    public
    view
    override
    returns (uint256 availableLiquidity, address underlying, uint256 currentFlow)
{
    (,, bool isTranched) = IAERC6909(aToken).getIdForUnderlying(asset);


    if (isTranched) {
        IFlowLimiter flowLimiter =
            IFlowLimiter(IMiniPoolAddressesProvider(_addressProvider).getFlowLimiter());
        underlying = IAToken(asset).UNDERLYING_ASSET_ADDRESS();
        address minipool = IAERC6909(aToken).getMinipoolAddress();
        currentFlow = flowLimiter.currentFlow(underlying, minipool);


          availableLiquidity = IERC20(asset).balanceOf(aToken) // <<<
              + IAToken(asset).convertToShares(flowLimiter.getFlowLimit(underlying, minipool)) //
              ↪  <<<
              - IAToken(asset).convertToShares(currentFlow); // <<<
    } else {
        availableLiquidity = IERC20(asset).balanceOf(aToken);
    }
}
```

However this leads to underestimating minipool utilization, and in-turn:

- If there is no "flow", underestimating liquidity rate.

  Indeed liquidity rate is derived from borrow rate by applying utilization factor, because borrow interest is shared proportionally to all available liquidity. However in this case "available flow" does not earn from this interest accrual, so no need to include it.

  - BasePiReserveRateStrategy.sol#L297-L306:

```
function getLiquidityRate(
    uint256 currentVariableBorrowRate,
    uint256 utilizationRate,
    uint256 reserveFactor
) internal pure returns (uint256) {
    return currentVariableBorrowRate.rayMul(utilizationRate).percentMul(
        PercentageMath.PERCENTAGE_FACTOR - reserveFactor
    );
}
```

- If there is "flow" (i.e minipool is borrowing from main), overestimating borrow rate.

  In this case a minimum liquidity rate is forced on the minipool so it is able to repay its debt eventually. In this case borrow rate is scaled by a factor of `1/utilization` in order to provide the liquidity rate to lenders. However as stated above, "available flow" does not earn interest so it leads to overcharging borrowers of minipool.

– MiniPoolPiReserveInterestRateStrategy.sol#197-L223:

```
    if (currentFlow != 0) {
        //@audit minliquidity rate formula
        ...

        // `&& utilizationRate != 0` to avoid 0 division. It's safe since the minipool flow is
        // always owed to a user. Since the debt is repaid as soon as possible if
        // `utilizationRate != 0` then `currentFlow == 0` by the end of the transaction.
        if (currentLiquidityRate < minLiquidityRate && utilizationRate != 0) {
            currentLiquidityRate = minLiquidityRate;
            currentVariableBorrowRate = currentLiquidityRate.rayDiv( // <<<
                utilizationRate.percentMul(PercentageMath.PERCENTAGE_FACTOR - reserveFactor)
                ↪  // <<<
            ); // <<<
        }
    }
```

Recommendation: Please consider removing available flow from available liquidity calculation:

- MiniPoolPiReserveInterestRateStrategy.sol#L91-L112:

```
function getAvailableLiquidity(address asset, address aToken)
    public
    view
    override
    returns (uint256 availableLiquidity, address underlying, uint256 currentFlow)
{
    (,, bool isTranched) = IAERC6909(aToken).getIdForUnderlying(asset);


    if (isTranched) {
        IFlowLimiter flowLimiter =
            IFlowLimiter(IMiniPoolAddressesProvider(_addressProvider).getFlowLimiter());
        underlying = IAToken(asset).UNDERLYING_ASSET_ADDRESS();
        address minipool = IAERC6909(aToken).getMinipoolAddress();
        currentFlow = flowLimiter.currentFlow(underlying, minipool);


-        availableLiquidity = IERC20(asset).balanceOf(aToken)
-            + IAToken(asset).convertToShares(flowLimiter.getFlowLimit(underlying, minipool))
-            - IAToken(asset).convertToShares(currentFlow);
+        availableLiquidity = IERC20(asset).balanceOf(aToken);
    } else {
        availableLiquidity = IERC20(asset).balanceOf(aToken);
    }
}
```

Please note that it would have the effect of keeping utilisation of minipool at 100% while bootstrapping liquidity (no organic liquidity provided to minipool yet). But liquidity rate would rise alongside borrow rate, and would attract lenders to supply to the minipool.

Cod3x: Fixed in commit 6c7bf89e.

Spearbit: Fix verified.

5.2.8 Using the index for `rewardTokens` could result in incorrect reward token transfers in `RewardForwarder` contract

Severity: Medium Risk

Context: RewardForwarder.sol#L68-L70

Description: In the `RewardForwarder` contract, rewards are distributed based on an index of reward tokens retrieved by the `RewardsController` as shown below:

```
function claimRewardsFor(address claimee, address token) public returns (uint256[] memory) {
    require(isRegisteredClaimee[claimee], "Not registered");
    address[] memory assets = new address[](1);
    assets[0] = token;
    (address[] memory rewardTokens_, uint256[] memory claimedAmounts_) = // <<<
        rewardsController.claimAllRewardsOnBehalf(assets, claimee, address(this));
    for (uint256 i = 0; i < rewardTokens_.length; i++) {
        claimedRewards[claimee][token][i] += claimedAmounts_[i]; // <<<
    }
    return claimedAmounts_;
}
```

The rewards are then claimed using the `rewardTokens` index to transfer tokens to the forwarder:

```
function forwardRewards(address claimee, address token, uint256 rewardTokenIndex) public {
    address rewardToken = rewardTokens[rewardTokenIndex]; // <<<
    uint256 amount = claimedRewards[claimee][token][rewardTokenIndex];
    require(amount != 0, "No rewards to forward");
    claimedRewards[claimee][token][rewardTokenIndex] = 0; // <<<
    address forwarder = forwarders[claimee][rewardTokenIndex];
    require(forwarder != address(0), "No forwarder set");
    IERC20(rewardToken).transfer(forwarder, amount); // <<<
}
```

A mismatch can occur when the `rewardTokens` arrays differ between contracts. For example, if `RewardForwarder` has `[TokenA]` while `RewardController` has `[TokenB]`, the `claimRewardsFor` function will record claimed amounts for TokenB, but `forwardRewards` will transfer TokenA instead.

Additionally, updating the `rewardTokens` array in `RewardForwarder` can lead to incorrect token distributions. For instance, if `rewardTokens` initially contains `[TokenA]`, the `rewardToken` will use the index 0, and if a user has `claimedRewards[claimee][token][0] = 100`, the function will transfer 100 TokenA. However, if `rewardTokens` is updated to `[TokenB]`, the same amount will be transferred in TokenB rather than TokenA.

Recommendation: Consider updating the function to use the token address instead of the index when claiming the tokens.

Cod3x: Fixed in commit e44fb3fe.

Spearbit: Fix verified.

5.2.9 `vars.availableLiquidity` is not capped by the reserve's total managed assets

Severity: Medium Risk

Context: MiniPoolDefaultReserveInterestRate.sol#L147-L149), MiniPoolPiReserveInterestRateStrategy.sol#L106-L108

Description: The following formual assumes there is enough liquidity in the corresponding tranched reserve in the lending pool to cover the unused flow (which might not be the case).

```
  vars.availableLiquidity = IERC20(asset).balanceOf(aToken)
+ IAToken(asset).convertToShares(flowLimiter.getFlowLimit(vars.underlying, minipool))
- IAToken(asset).convertToShares(vars.currentFlow);
```

Let's define `vars.flowLiquidity` as:

```
  vars.flowLiquidity = IAToken(asset).convertToShares(flowLimiter.getFlowLimit(vars.underlying,
  ↪  minipool))
- IAToken(asset).convertToShares(vars.currentFlow)
```

then this `vars.flowLiquidity` needs to be capped by the liquidity available in the corresponding reserve in the lending pool:

```
vars.flowLiquidity = min(
   vars.flowLiquidity,
   IAToken(asset).convertToShares(
      IAToken(asset).ATOKEN_ADDRESS().getTotalManagedAssets()
   )
);
```

Recommendation: Based on the above one should put a cap on the available flow and modify the formula used for `vars.availableLiquidity` as:

```
vars.flowLiquidity = IAToken(asset).convertToShares(flowLimiter.getFlowLimit(vars.underlying, minipool))
   - IAToken(asset).convertToShares(vars.currentFlow);

vars.flowLiquidity = min(
   vars.flowLiquidity,
   IAToken(asset).convertToShares(
      IAToken(asset).ATOKEN_ADDRESS().getTotalManagedAssets()
   )
);

vars.availableLiquidity = IERC20(asset).balanceOf(aToken) + vars.flowLiquidity
```

Cod3x: The unused flow has been removed from `vars.availableLiquidity` in commit 6c7bf89e and thus this issue does not apply anymore.

Spearbit: Verified.

5.2.10 Reserves can be re-added

Severity: Medium Risk

Context: LendingPool.sol#L777-L791, MiniPool.sol#L708-L721

Description: If the reserve for that specific `asset` and `reserveType` has already been added, `revert` instead. If we don't some of the important parameters can be rewritten and also the liquidity and borrow indices would be reset to one `RAY`.

Recommendation: It would be best to revert `reserveAlreadyAdded` is `true`.

Cod3x: Fixed in commit 4fb61a93.

Spearbit: Fix verified.

5.2.11 Minipool owner can create unliquidatable loan, imposing bad debt on main lending pool

Severity: Medium Risk

Context: MiniPoolConfigurator.sol#L481-L490

Description: In the scenario where minipool owner is not the same entity as the main pool admin, and the minipool has some borrowing capacity from the main lending pool (available flow), the minipool owner can create an unliquidatable loan, forcing bad debt upon the main lending pool.

> Note that initially main lending pool owner and minipools owner should be the same entity. The risk reported here would be impactful with decentralization of minipool ownership in the future.

To achieve this, the minipool owner can simply deactivate the reserve associated with the collateral of his loan. This is possible due to a flawed `_checkNoLiquidity` for deactivating reserves:

- MiniPoolConfigurator.sol#L481-L490:

```
function _checkNoLiquidity(address asset, IMiniPool pool) internal view {
    DataTypes.MiniPoolReserveData memory reserveData = pool.getReserveData(asset);

    uint256 availableLiquidity = IERC20Detailed(asset).balanceOf(reserveData.aTokenAddress); //
    ↪  <<<

    require(
        availableLiquidity == 0 && reserveData.currentLiquidityRate == 0,
        Errors.LPC_RESERVE_LIQUIDITY_NOT_0
    );
}
```

Scenario: Let's examine the concrete way of creating an unliquidatable loan:

- Parameters:

Parameter	Value
Collateral	aWETH
Debt	aUSDC
Available flow	500k aUSDC
aUSDC Liquidity	0
aWETH LTV	1.2

- Steps:
  - Alice the minipool owner deposits 600k$ of WETH into the minipool and borrows 500k USDC, using all available flow.
  - Alice flashloans all of WETH out of erc6909 market, enabling the deactivation of reserve because `_checkNoLiquidity` allows it.

Recommendation: A more robust check could be made on supply of asset:

- MiniPoolConfigurator.sol#L481-L490:

```
function _checkNoLiquidity(address asset, IMiniPool pool) internal view {
    DataTypes.MiniPoolReserveData memory reserveData = pool.getReserveData(asset);

-    uint256 availableLiquidity = IERC20Detailed(asset).balanceOf(reserveData.aTokenAddress);
+    uint256 availableLiquidity = IERC6909(reserveData.aTokenAddress).scaledTotalSupply();

    require(
        availableLiquidity == 0 && reserveData.currentLiquidityRate == 0,
        Errors.LPC_RESERVE_LIQUIDITY_NOT_0
    );
}
```

Cod3x: Fixed in commit df0b9abf.

Spearbit: Fix verified.

5.2.12 `getCurrentInterestRates()` does not calculate the interests correctly

Severity: Medium Risk

Context: PiReserveInterestRateStrategy.sol#L97-L100, MiniPoolPiReserveInterestRateStrategy.sol#L131-L133

Description: The current borrow rate in `getCurrentInterestRates()` is calculated as:

```
// borrow rate
uint256 currentVariableBorrowRate =
    transferFunction(getControllerError(getNormalizedError(utilizationRate)));
```

This does not take into the consideration the following storage update for `_errI` which is used in `getControllerError`:

```
_errI += int256(_ki).rayMulInt(err * int256(block.timestamp - _lastTimestamp));
if (_errI < _maxErrIAmp) _errI = _maxErrIAmp; // Limit _errI negative accumulation.
```

The NatSpec for `getCurrentInterestRates()` mentions:

```
/**
 * @notice The view version of `calculateInterestRates()`.
 * @dev Returns the current interest rates without modifying state.
 ...
 */
```

Recommendation:

1. Make sure the current value of `_errI` is used in the derivation of `currentVariableBorrowRate`.
2. Make sure the `availableLiquidity` for mini pools follows the same formual used `calculateInterestRates()` which currently includes taking into account the unused flows for tranched reserves in the mini pool. Use:

```
int256 err = getNormalizedError(utilizationRate);
int256 currentControllerError = (
    _errI
    + int256(_ki).rayMulInt(err * int256(block.timestamp - _lastTimestamp))
);

if (currentControllerError < _maxErrIAmp) currentControllerError = _maxErrIAmp; // Limit _errI
↪  negative accumulation.
currentControllerError += int256(_kp).rayMulInt(err);
// borrow rate
uint256 currentVariableBorrowRate =
    transferFunction(currentControllerError);
```

The above logic can be refactored by a utility function that can be shared between `_calculateInterestRates` and `getCurrentInterestRates`. `getAvailableLiquidity()`.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.2.13 `variableBorrowIndex` is only updated in `_updateIndexes` of the mini pool if `currentLiquidityRate` is non-zero

Severity: Medium Risk

Context: MiniPoolReserveLogic.sol#L320-L340

Description: The second `if` block is nested in this context, although it is not nested in lending pool's `ReserveLogic` library. (in `Aave v2` it is nested):

```
if (currentLiquidityRate != 0) {
    // ...
    if (scaledVariableDebt != 0) {
        // ...
    }
}
```

But it is possible that `scaledVariableDebt` is non-zero even when `currentLiquidityRate` is `0` during to rounding errors or for example when both the `Cod3x` and reserve's owner's reserve factors sum up to `100%`.

Recommendation: Make sure the `if` blocks are not-nested and let the index updates happen separately:

```
if (currentLiquidityRate != 0) {
    // ...
}

if (scaledVariableDebt != 0) {
    // ...
}
```

Cod3x: Fixed in commit 5ddd6f4b.

Spearbit: Fix verified.

5.2.14 Mini pool reserves for unique tokens can be reinitialised

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Mini pool reserves can be reinitialised. This is due to the following factors:

1. For unique tokens one can add them to `ATokenERC6909` multiple times but with different incremental IDs.
2. In `MiniPoolReserveLogic` we have:

```
function init(
    DataTypes.MiniPoolReserveData storage reserve,
    address asset,
    IAERC6909 aTokenAddress,
    uint256 aTokenID,
    uint256 variableDebtTokenID,
    address interestRateStrategyAddress
) internal {
    require(
        aTokenAddress.getUnderlyingAsset(aTokenID) == asset, // this just checks to make sure the
        ↪  `asset` has been added in `aTokenAddress`
        Errors.RL_RESERVE_ALREADY_INITIALIZED
    );

    reserve.liquidityIndex = uint128(WadRayMath.ray());
    reserve.variableBorrowIndex = uint128(WadRayMath.ray());
    reserve.aTokenAddress = address(aTokenAddress);
    reserve.aTokenID = aTokenID;
    reserve.variableDebtTokenID = variableDebtTokenID;
    reserve.interestRateStrategyAddress = interestRateStrategyAddress;
}
```

This will cause.

1. `liquidityIndex` and `variableBorrowIndex` be reset.
2. `aTokenID` and `variableDebtTokenID` be updated and thus all user accounting will be lost.

3. `interestRateStrategyAddress` be updated which for stateful IRMs can cause an issue.

Recommendation:

1. Make sure there is only one reserve per unique token just like the non-rebasing AToken of the main lending pool or...
2. The mini pool needs to be restructured so that adding a unique token multiple times would not override the reserve of the perviously added unique token.

Cod3x: Fixed in commit b46b15a3.

Spearbit: Fix verified.

5.2.15 Pi interest rate model is manipulatable due to current balances used

Severity: Medium Risk

Context: BasePiReserveRateStrategy.sol#L253

Description: The interest rate is controlled the utilisation factor:

- BasePiReserveRateStrategy.sol#L234-L265:

```
function _calculateInterestRates(
    address,
    uint256 availableLiquidity,
    uint256 totalVariableDebt,
    uint256 reserveFactor
) internal returns (uint256, uint256, uint256) {
    uint256 utilizationRate = totalVariableDebt == 0
        ? 0
        : totalVariableDebt.rayDiv(availableLiquidity + totalVariableDebt);


    // If no borrowers we reset the strategy
    if (utilizationRate == 0) {
        _errI = 0;
        _lastTimestamp = block.timestamp;
        return (0, 0, 0);
    }


    // PID state update
    int256 err = getNormalizedError(utilizationRate);
    _errI += int256(_ki).rayMulInt(err * int256(block.timestamp - _lastTimestamp)); // <<<
    if (_errI < _maxErrIAmp) _errI = _maxErrIAmp; // Limit _errI negative accumulation.
    _lastTimestamp = block.timestamp;
    int256 controllerErr = getControllerError(err);
    uint256 currentVariableBorrowRate = transferFunction(controllerErr);
    uint256 currentLiquidityRate =
        getLiquidityRate(currentVariableBorrowRate, utilizationRate, reserveFactor);


    emit PidLog(
        utilizationRate, currentLiquidityRate, currentVariableBorrowRate, err, controllerErr
    );
    return (currentLiquidityRate, currentVariableBorrowRate, utilizationRate);
}
```

Unfortunately since instant values for available liquidity and debt are used (and `_errI` is only updated once per block), it is possible for participants to manipulate the direction of the rate at their advantage by using a flash loan or a borrow:

- Borrower side: The borrower can first use a flash loan to deposit into the pool, then withdraw and finally borrow (`_errI` won't be updated for the last two operations).
- Lender side: The lender can first borrow, increasing `_errI`, and then repay same amount (repaying won't change `_errI` in the same block).

Recommendation: Use previous `availableLiquidity` and `totalVariableDebt` to update `_errI`.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.3 Low Risk

5.3.1 Events related issues

Severity: Low Risk

Context: LendingPoolAddressesProvider.sol#L197-L199, LendingPoolAddressesProvider.sol#L213-L215, MiniPoolAddressProvider.sol#L280, MiniPoolAddressProvider.sol#L292, MiniPoolAddressProvider.sol#L326, MiniPoolAddressProvider.sol#L354, MiniPoolAddressProvider.sol#L365, LendingPool.sol#L810

Description/Recommendation:

- LendingPoolAddressesProvider.sol#L197-L199, LendingPoolAddressesProvider.sol#L213-L215, AToken.sol#L506-L510, AToken.sol#L610-L679, VariableDebtToken.sol#L392-L395, ATokenERC6909.sol#L147-L150, RewardForwarder.sol#L46-L147, RewardsController.sol#L65-L76, LendingPool.sol#L809-L811, BorrowLogic.sol#L441, MiniPoolBorrowLogic.sol#L337: Events missing.
- MiniPoolAddressProvider.sol#L280, MiniPoolAddressProvider.sol#L292, MiniPoolAddressProvider.sol#L365: Also emit `miniPoolId` in these events.
- MiniPoolAddressProvider.sol#L354: Also emit `miniPool` and `asset` in `FlowLimitUpdated`.
- MiniPoolAddressProvider.sol#L326: Emitting `PoolAdminSet` is missing.
- AToken.sol#L158-L168: `_aTokenWrapper` is missing from `Initialized`.
- BasePiReserveRateStrategy.sol: Event emission is missing from the setter functions in this contract.
- ATokenERC6909.sol#L30: Minting endpoints do not emit the `Mint` event when compared to `AToken` counterpart.
- BorrowLogic.sol#L308: In `IVariableDebtToken(reserve.variableDebtTokenAddress).mint` the `onBehalfOf` input parameter is set to `params.miniPoolAddress`, but for the `Borrow` event the `onBehalfOf` parameter is set to `address(flowLimiter)`.

There are more endpoints that don't emit events, please consider what is needed for on and off-chain agents and provide reverent event emission based on the project needs..

Cod3x: Fixed in commit 2d70fcc7.

Spearbit: Fix verified.

5.3.2 `poolIdCheck(miniPoolId)` modifier is missing

Severity: Low Risk

Context: MiniPoolAddressProvider.sol#L349, MiniPoolAddressProvider.sol#L362

Description/Recommendation: MiniPoolAddressProvider.sol#L349, MiniPoolAddressProvider.sol#L362: `poolIdCheck(miniPoolId)` modifier is missing.

Cod3x: Fixed in commit 725d64c2.

Spearbit: Fix verified.

5.3.3 `DOMAIN_SEPARATOR` does not adjust to the changes in `block.chainid` or `address(this)`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: AToken.sol#L134-L142: If a fork happens `DOMAIN_SEPARATOR` does not get recalculated automatically instead one needs to reinitialise the proxy again.

Recommendation: If `block.chainid` or `address(this)` changes make sure to recalculate `DOMAIN_SEPARATOR`.

Cod3x: Fixed in commit f3ca85f8.

Spearbit: The fix only take into consideration the change in `block.chainid` to recalculate the domain separator and not `address(this)`.

5.3.4 `transfer...` logic for debt tokens in `ATokenERC6906` does not incorporate indexes

Severity: Low Risk

Context: ATokenERC6909.sol#L183, ATokenERC6909.sol#L216

Description: It is true that only the mini pool can transfer debt tokens between users (as is enforced in the before token transfer hook) and currently it doesn't use this logic, but the transfer logic implemented for the debt token transfer is incorrect as it does not incorporate the debt index.

Recommendation: The transfer logic should follow this logic:

```
} else {
    address underlyingAsset = _underlyingAssetAddresses[id];
    uint256 index = POOL.getReserveNormalizedVariableDebt(underlyingAsset);
    // ...
    super.transferFrom(from, to, id, amount.rayDiv(index));
    // ...
}
```

Cod3x: Fixed in commit 725d64c2.

Spearbit: Fix verified.

5.3.5 `RewardsController` incorrect tracking the miniPools after updating `addressesProvider`

Severity: Low Risk

Context: RewardsController.sol#L155-L171

Description: The `_addressesProvider` variable in `RewardsController` can be updated through `setMiniPoolAddressesProvider`. If changed, the tracking of miniPools may become incorrect since the new address provider could have different miniPools and a different total count. This mismatch between new and old miniPools could break the `_isAtokenERC6909` and `_isMiniPool` arrays, possibly breaking the `handleAction` function.

Recommendation: Consider reseting the tracking of the miniPools when updating the `addressesProvider` variable.

Cod3x: Fixed in commit 99bbbf93.

Spearbit: Fix verified.

5.3.6 Boundary check is missing when setting `_optimalUtilizationRate` in the constructor

Severity: Low Risk

Context: BasePiReserveRateStrategy.sol#L98

Description: this check is missing in the context:

```
if (optimalUtilizationRate >= uint256(RAY)) {
    revert(Errors.IR_UO_GREATER_THAN_RAY);
}
```

best to use internal setter functions that refactors all the necessary logic.

Recommendation: Instead of setting the storage parameter manually call define and call `_setOptimalUtilizationRate(optimalUtilizationRate)` where all the necessary logic is defined in this setter function including the boundary check and event emission.

Cod3x: Fixed in commit 1eb7b435.

Spearbit: Fix verified.

5.3.7 `_setAssetsSources` does not compare all lengths and is not emitting all parameters

Severity: Low Risk

Context: Oracle.sol#L105-L109

Description:

1. `timeouts` length has not been compared to `assets.length`.
2. `timeouts[i]` does not get emitted in the event `AssetSourceUpdated`.

Recommendation: Make sure all lengths are compared for the arrays supplied to `_setAssetsSources` and the event include all parameters.

Cod3x: Fixed in commit 1eb7b435.

Spearbit: Fix verified.

5.3.8 Unchecked zero address for profit handler can lead to loss of profits

Severity: Low Risk

Context: AToken.sol#L597

Description: During a rebalance (AToken rehypothetication), profits made from a strategy are evaluated and sent to a `_profitHandler` address.

- AToken.sol#L593-L601:

```
if (profit != 0) {
    // Profit is ultimately (coll at hand) + (coll allocated to yield generator) - (recorded
    ↪  total coll Amount in pool).
    profit =
        (IERC20(_underlyingAsset).balanceOf(address(this)) + _farmingBal) - _underlyingAmount;
    if (profit != 0) {
        // Distribute to profitHandler.
        IERC20(_underlyingAsset).safeTransfer(_profitHandler, profit);
    }
}
```

However it is not checked that `_profitHandler` is initialized. Since this member is not set during initialization, but only by its specific setter, it could be unset. This would lead to the burning of the accumulated profit.

- AToken.sol#L635-L643:

```
/**
 * @dev Sets the profit handler address.
 * @param profitHandler The new profit handler address.
 */
function setProfitHandler(address profitHandler) external override onlyLendingPool {
    require(profitHandler != address(0), Errors.AT_INVALID_ADDRESS);
    require(address(_vault) != address(0), Errors.AT_VAULT_NOT_INITIALIZED);
    _profitHandler = profitHandler;
}
```

Recommendation: Only send the profit amount if `_profitHandler` is set:

- AToken.sol#L593-L601:

```
  if (profit != 0) {
      // Profit is ultimately (coll at hand) + (coll allocated to yield generator) - (recorded total
      ↪  coll Amount in pool).
      profit =
          (IERC20(_underlyingAsset).balanceOf(address(this)) + _farmingBal) - _underlyingAmount;
-     if (profit != 0) {
+     if (profit != 0 && _profitHandler != address(0)) {
          // Distribute to profitHandler.
          IERC20(_underlyingAsset).safeTransfer(_profitHandler, profit);
      }
  }
```

Cod3x: Fixed in commits 1eb7b435 and 9c76b33c.

Spearbit: Fix verified.

5.3.9 `WadRayMath.rayPowerInt` should return `RAYint` instead of 1 when `exponent == 0`

Severity: Low Risk

Context: WadRayMath.sol#L192-L194

Description: `WadRayMath.rayPowerInt` is supposed to compute the number `base.pow(exponent)` in the `RAY` precision. The result should be 1 when `exponent == 0`, however the function `rayPowerInt` returns `1` instead of `RAYint` (which is 1 in `RAY` precision).

- WadRayMath.sol#L191-L194:

```
function rayPowerInt(int256 base, uint256 exponent) internal pure returns (int256) {
    if (exponent == 0) {
        return 1; // <<<
    }
    // ...
}
```

Please note that this function is nowhere used with exponent value 0, which is why the severity is marked as low for the current codebase.

Recommendation: Return `RAYint` instead:

- WadRayMath.sol#L191-L194:

```
   function rayPowerInt(int256 base, uint256 exponent) internal pure returns (int256) {
       if (exponent == 0) {
-          return 1;
+          return RAYint;
       }
```

Cod3x: Fixed in commit 1eb7b435.

Spearbit: Fix verified.

5.3.10 First added reserve asset cannot be added again with the other reserve type

Severity: Low Risk

Context: LendingPool.sol#L777-L791

Description: Reserves list has been updated to contain an address and a reserve type (rehypothetication enabled/disabled). However in the condition checking if a reserve has been added (in the first token special case), only asset is checked. This would prevent adding the first asset again, but with the other reserve type.

Recommendation:

```
  function _addReserveToList(address asset, bool reserveType) internal {
      uint256 reservesCount = _reservesCount;

      require(reservesCount < _maxNumberOfReserves, Errors.LP_NO_MORE_RESERVES_ALLOWED);

      bool reserveAlreadyAdded =
-         _reserves[asset][reserveType].id != 0 || _reservesList[0].asset == asset;
+         _reserves[asset][reserveType].id != 0 ||
+             (_reservesList[0].asset == asset && _reservesList[0].reserveType == reserveType);

      if (!reserveAlreadyAdded) {
          _reserves[asset][reserveType].id = uint8(reservesCount);
          _reservesList[reservesCount] = DataTypes.ReserveReference(asset, reserveType);

          _reservesCount = reservesCount + 1;
      }
  }
```

Cod3x: Fixed in commit 1eb7b435.

Spearbit: Fix verified.

5.3.11 Division by zero in `calculateInterestRates` due to `reserveFactor` being 100%

Severity: Low Risk

Context: MiniPoolPiReserveInterestRateStrategy.sol#L217

Description: Aggregate (`Cod3x` and `MinipoolOwner`) `reserveFactor` can be 100%, and in that case a divide by zero could dos interest rate update in `MiniPoolPiReserveInterestRateStrategy.calculateInterestRates`.

Recommendation: Check that `PercentageMath.PERCENTAGE_FACTOR - reserveFactor` is not zero:

```
- if (currentLiquidityRate < minLiquidityRate && utilizationRate != 0) {
+ if (currentLiquidityRate < minLiquidityRate &&
+       utilizationRate != 0 &&
+       (PercentageMath.PERCENTAGE_FACTOR - reserveFactor != 0)) {
      currentLiquidityRate = minLiquidityRate;
      currentVariableBorrowRate = currentLiquidityRate.rayDiv(
          utilizationRate.percentMul(PercentageMath.PERCENTAGE_FACTOR - reserveFactor)
      );
  }
```

Cod3x: Fixed in commit 1eb7b435.

Spearbit: Perhaps this scenario should be handled differently. Since we want to always make sure that the mini pool can repay its borrowing position in the main lending pool when there is a non-zero flow.

5.3.12 Make sure `vars.amountReceived` is at least `amount - vars.availableLiquidity`

Severity: Low Risk

Context: MiniPool.sol#L233

Description/Recommendation: A check can be added in this context to make sure `vars.amountReceived` is at least `amount - vars.availableLiquidity` not because of rounding issue but because future implementations might not guarantee certain assumptions so it is best to check this invariant here.

Cod3x: `assert(vars.amountReceived >= amount - vars.availableLiquidity);` has been added in commit 1eb7b435.

Spearbit: Fix verified.

5.3.13 Inconstant health factor inequality boundary checks

Severity: Low Risk

Context: ValidationLogic.sol#L201, MiniPoolValidationLogic.sol#L202-L205

Description: In other checks regarding healthy positions the inequality is not strict, but in this context a strict inequality is checked.

```
require(
    vars.healthFactor > /*...*/PoolGenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
    Errors.VL_HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD
);
```

Recommendation: Make sure all the inequality checks are consistent or provide a comment/reason as to why it is different in this case.

Cod3x: Fixed in commit 1eb7b435.

Spearbit: Fix verified.

5.3.14 `getAssetPrice` assumes that the quote token decimals and oracle precisions match

Severity: Low Risk

Context: Oracle.sol#L147-L148.

Description: In this context we have:

```
(uint80 roundId, int256 price, uint256 startedAt, uint256 timestamp,) =
   IChainlinkAggregator(source).latestRoundData();
```

It is important to note that the quote currency's decimals and the oracle precision might not be equal in general. Although of the case `ETH` all `X / ETH` oracles have the `18` precision according to the docs. In general we have:

$$\texttt{price} = \frac{[Q]/10^{d_Q}}{[B]/10^{d_B}} \cdot 10^{d_O}$$

when the quote's currency decimals equals to the oracle precision this formulas simplifies to:

$$\texttt{price} = \frac{[Q]}{[B]/10^{d_B}}$$

which is the assumption used in this codebase.

Recommendation: Either document above or use the more general formula throughout the codebase.

Cod3x: Governance is responsible for ensuring that all Oracle aggregators return the correct number of decimals. A documentation item has been added.

Spearbit: Acknowledged.

5.3.15 Analysis of `balanceDecreaseAllowed` when there is at least one reserve borrowed but the `vars.totalDebtInETH` is `0`

Severity: Low Risk

Context: MiniPoolGenericLogic.sol#L79-L84, GenericLogic.sol#L97-L99

Description:

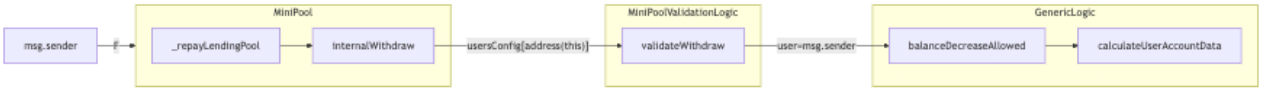
1. Mini Pool:

   In the usual flows the `return` statement in this context cannot be reached, since we have already checked `userConfig.isBorrowingAny()` and so if we get to this `if` block we know that `userConfig` is borrowing at least one reserve.

   ```
   if (!userConfig.isBorrowingAny() || !userConfig.isUsingAsCollateral(reserves[asset].id)) {
       return true;
   }

   // ...

   (vars.totalCollateralInETH, vars.totalDebtInETH,, vars.avgLiquidationThreshold,) =
   calculateUserAccountData(user, reserves, userConfig, reservesList, reservesCount, oracle);

   if (vars.totalDebtInETH == 0) {
       return true; // <--- the `return` statement in the question
   }
   ```

   The question arrises as to when can this happen, aka for a user who has at least one borrowing reserve but with the `vars.totalDebtInETH` equal to `0`.

   1. `vars.totalDebtInETH` could be `0` since in its calculation we underestimate / round down and thus possible to reach this return statement due to this rounding direction error or.
   2. That the mini pool is borrowing some reserves (`userConfig` corresponds to the minipool in the `internalWithdraw` flow but the `user=msg.sender` is not borrowing any of those).
   3. When one calls `calculateUserAccountData(user, ..., userConfig, ...)` in the mini pool `deposit` (or in general `F`) flow, `user` would be `msg.sender` but `userConfig` would be `usersConfig[address(this)]` which is the user config for the mini pool.

   where `F` can be `deposit`, `repay`, or `liquidationCall`.

   The cases `2.` and `3.` currently could not happen since there are no reserves marked as borrowed for the mini pool in the mini pool itself and thus the call to `balanceDecreaseAllowed(...)` returns early in these two cases.

2. Lending Pool: This is the same as the mini pool case regarding the first case where:

   1. `vars.totalDebtInETH` could be `0` since in its calculation we underestimate / round down and thus possible to reach this return statement due to this rounding direction error.

Recommendation: In general one should have the following invariant that when `userConfig.isBorrowingAny()` is `true` the `vars.totalDebtInETH` should be non-zero. This can be guaranteed if in all calculations to

derive `vars.totalDebtInETH` one chooses a rounding direction that would favour over-estimating the `vars.totalDebtInETH`.

Cod3x: Fixed in commit 1ee6d04b.

Spearbit: Fix verified.

5.3.16 The word `principal` is used with different connotations leading to confusion

Severity: Low Risk

Context: ReserveLogic.sol#L259, ValidationLogic.sol#L330, ValidationLogic.sol#L335, ValidationLogic.sol#L342, ValidationLogic.sol#L349, MiniPoolLiquidationLogic.sol#L48, MiniPoolLiquidationLogic.sol#L57, MiniPoolReserveLogic.sol#L268, MiniPoolValidationLogic.sol#L335, MiniPoolValidationLogic.sol#L340, MiniPoolValidationLogic.sol#L347, MiniPoolValidationLogic.sol#L354, RewardsDistributor.sol#L457, RewardsDistributor.sol#L458, RewardsDistributor.sol#L465, RewardsDistributor.sol#L470, RewardsDistributor6909.sol#L485, RewardsDistributor6909.sol#L486, RewardsDistributor6909.sol#L493, RewardsDistributor6909.sol#L498, AToken.sol#L267, VariableDebtToken.sol#L278, VariableDebtToken.sol#L305, VariableDebtToken.sol#L307, VariableDebtToken.sol#L308

Description:

1. In ReserveLogic, MiniPoolReserveLogic we have:

   ```
   // Calculate the last principal variable debt.
   vars.previousVariableDebt = scaledVariableDebt.rayMul(previousVariableBorrowIndex);
   ```

   and so `principal` means the actual debt principal plus the accrued interest.

2. In liquidation flows the word `principal` is used in conjugation with `reserve` to mean the debt or borrow reserve.

3. In RewardsDistributor and RewardsDistributor6909 we have:

   ```
   /**
    * @dev Calculates rewards based on principal balance and index difference.
    * @param principalUserBalance User's principal balance.
    * @param reserveIndex Current reserve index.
    * @param userIndex User's stored index.
    * @param decimals Number of decimals.
    * @return The calculated reward amount.
    */
   function _getRewards(
       uint256 principalUserBalance,
       uint256 reserveIndex,
       uint256 userIndex,
       uint8 decimals
   ) internal pure returns (uint256) {
       return (principalUserBalance * (reserveIndex - userIndex)) / 10 ** decimals;
   }
   ```

   Here `principal` based on the analysis of the input (avoiding the mix of scaling related issues) is supposed to be the amount of `shares` which is actually not the initial principal deposited or borrowed by the user.

4. AToken: Here the word `principal` actually matches with what we would consider as principal:

   ```
   // the balance of the user: principal balance + interest generated by the principal.
   super.balanceOf(user).rayMul(
       _pool.getReserveNormalizedIncome(_underlyingAsset, RESERVE_TYPE)
   )
   ```

5. In VariableDebtToken.sol#L277-L284, VariableDebtToken.sol#L304-L317: The word `principal` refers to the shares and total shares in this token.

Recommendation: In most cases the comments or variable names need to be updated to reflect the actual value computed or stored, but in the case of `3.` the confusion can result in computing incorrect reward amounts for the user. The question comes into mind that whether the user needs to be awards based on:

- The principal deposited or borrowed (the true meaning of the word) or...
- The shares of the `aToken` or `debtToken` the user holds.

If it is the first point then the reward distribution flow needs to be updated and the tokens would also need to store principal values along the shares to be able to provide them to the incentive controllers.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 `_miniPoolCount` can be provided as parameters to `_initMiniPool` and `_initATokenPool`

Severity: Gas Optimization

Context: MiniPoolAddressProvider.sol#L432-L440, MiniPoolAddressProvider.sol#L447-L454

Description/Recommendation: `_miniPoolCount` can be provided as parameters to `_initMiniPool` and `_initATokenPool` to avoid reading from storage multiple times.

Cod3x: Fixed in commit 4bbb4351.

Spearbit: Fix verified.

5.4.2 Instead of the linear search `_getMiniPoolId` introduce a reverse-mapping storage value

Severity: Gas Optimization

Context: MiniPoolAddressProvider.sol#L185-L192

Description: In `_getMiniPoolId` each pool is queried to find the one matching with `miniPool`. This function will be invoked in mini pool borrowing or incentivised token's `handleAction` and thus it would be best to be optimised.

Recommendation: Instead of the linear search `_getMiniPoolId` introduce a reverse-mapping storage value.

Cod3x: Fixed in commit 4bbb4351.

Spearbit: Fix verified.

5.4.3 `oldTotalSupply` can be reused in `IncentivizedERC6909`

Severity: Gas Optimization

Context: *(No context files were provided by the reviewer)*

Description/Recommendation: `oldTotalSupply` can be reused in IncentivizedERC6909:

```
_totalSupply[id] = oldTotalSupply +/- amt;
```

Cod3x: Fixed in commit 4bbb4351.

Spearbit: Fix verified.

5.4.4 `refreshMiniPoolData` can be optimised

Severity: Gas Optimization

Context: RewardsController.sol#L157-L169

Description/Recommendation: Cache `_addressesProvider` and also `_totalTrackedMiniPools` can be updated after the loop in this context.

Cod3x: Fixed in commit 4bbb4351.

Spearbit: Fix verified.

5.4.5 `MiniPoolPiReserveInterestRateStrategy.getAvailableLiquidity` can be optimised

Severity: Gas Optimization

Context: MiniPoolPiReserveInterestRateStrategy.sol#L99-L111

Description/Recommendation: `MiniPoolPiReserveInterestRateStrategy.getAvailableLiquidity` can be optimised:

```
availableLiquidity = IERC20(asset).balanceOf(aToken);

if (isTranched) {
    IFlowLimiter flowLimiter =
        IFlowLimiter(IMiniPoolAddressesProvider(_addressProvider).getFlowLimiter());
    underlying = IAToken(asset).UNDERLYING_ASSET_ADDRESS();
    address minipool = IAERC6909(aToken).getMinipoolAddress();
    currentFlow = flowLimiter.currentFlow(underlying, minipool);

    availableLiquidity +=
        IAToken(asset).convertToShares(flowLimiter.getFlowLimit(underlying, minipool))
        - IAToken(asset).convertToShares(currentFlow);
}
```

- `forge s --diff`:

```
testReservesForMiniPools(uint256) (gas: -20 (-0.000%))
testMinipoolZigturH5_2_1(uint256,uint256,uint256) (gas: 6 (0.000%))
testMpProvider(uint256) (gas: -5 (-0.000%))
testBorrowRepayAndWithdrawWithFlow(uint256,uint256,uint256) (gas: -7 (-0.000%))
testErc6909Minting_DebtToken(uint256,uint256,uint256) (gas: -2 (-0.000%))
testWithdrawalsZeroDebt(uint256,uint256) (gas: 7 (0.001%))
testMiniPoolDeposits(uint256,uint256) (gas: -6 (-0.001%))
testMiniPoolDeposits(uint256,uint256) (gas: 7 (0.001%))
testMultipleUsersBorrowRepayAndWithdraw(uint256,uint256,uint256) (gas: -17 (-0.001%))
testTransferCollateral(uint256,uint256) (gas: 12 (0.001%))
testMiniPoolReserveFactors(uint256,uint256,uint256,uint256) (gas: 45 (0.001%))
testMiniPoolNormalBorrow(uint256,uint256,uint256) (gas: 26 (0.001%))
testWithdrawWhenBorrowed(uint256,uint256,uint256,uint256) (gas: 38 (0.001%))
testMiniPoolDeposits(uint256,uint256) (gas: 14 (0.001%))
testMiniPoolDeposits(uint256,uint256) (gas: -15 (-0.001%))
testMiniPoolNormalBorrow(uint256,uint256,uint256) (gas: -40 (-0.002%))
testMiniPoolReserveFactors(uint256,uint256,uint256,uint256) (gas: -82 (-0.002%))
testMiniPoolReserveFactors(uint256,uint256,uint256,uint256) (gas: -87 (-0.002%))
testSetUserUseReserveAsCollateral(uint256,uint256,uint256) (gas: -45 (-0.002%))
testMiniPoolReserveFactors(uint256,uint256,uint256,uint256) (gas: 108 (0.002%))
testPidMiniPool() (gas: 4025 (0.003%))
testReserveFactorPositiveInNormalBorrow(uint256,uint256,uint256,uint256) (gas: 84 (0.004%))
testErc6909Transfer_AToken(uint256,uint256,uint256) (gas: -35 (-0.004%))
testMiniPoolNormalBorrow(uint256,uint256,uint256) (gas: -148 (-0.006%))
testErc6909TransferOnLiquidation_AToken(uint256,uint256,uint256) (gas: -230 (-0.017%))
testMiniPoolNormalBorrow(uint256,uint256,uint256) (gas: -2833 (-0.110%))
testMiniPoolNormalBorrowRepay(uint256,uint256,uint256) (gas: -5134 (-0.175%))
testCannotWithdrawWhenBorrowedMaxLtv(uint256,uint256,uint256,uint256) (gas: 5767 (0.219%))
testErc6909TransferFrom_AToken(uint256,uint256,uint256) (gas: 13060 (0.484%))
testMiniPoolLiquidation(uint256,uint256,uint256,uint256) (gas: -38271 (-1.228%))
Overall gas change: -23778 (-0.000%)
```

The same recommendation can be applied to MiniPoolDefaultReserveInterestRate.sol#L141-L152.

Cod3x: Fixed in commit 4bbb4351.

Spearbit: Not quite relevant anymore since, the unused flow is not used anymore in the calculation of `availableLiquidity`. Although using this recommendation the `if/else` block has been simplified and also `availableLiquidity` calculation has been moved out and above this block.

5.4.6 Simplification in liquidation logic

Severity: Gas Optimization

Context: LiquidationLogic.sol#L213-L221

Description: The function`liquidationCall`contains legacy code from AAVE v2 that includes stable debt. Since the current code only uses variable debt, it can be simplified by removing the else statement in:

```
if (vars.userVariableDebt >= vars.actualDebtToLiquidate) {
    IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
        params.user, vars.actualDebtToLiquidate, debtReserve.variableBorrowIndex
    );
} else {
  IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
      params.user, vars.userVariableDebt, debtReserve.variableBorrowIndex
   );
}
```

Recommendation: Consider removing this unreachable code block:

```
- if (vars.userVariableDebt >= vars.actualDebtToLiquidate) {
      IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
          params.user, vars.actualDebtToLiquidate, debtReserve.variableBorrowIndex
      );
- } else {
-     IVariableDebtToken(debtReserve.variableDebtTokenAddress).burn(
-         params.user, vars.userVariableDebt, debtReserve.variableBorrowIndex
-     );
- }
```

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 Unused/unreachable/redundant/... code

Severity: Informational

Context: MiniPoolAddressProvider.sol#L158-L160, MiniPoolAddressProvider.sol#L406, MiniPoolAddressProvider.sol#L424, LendingPoolStorage.sol#L4-L8, LendingPoolStorage.sol#L19-L21, MiniPoolStorage.sol#L4-L8, MiniPoolStorage.sol#L21-L23

Description / Recommendation:

- MiniPoolStorage.sol#L4-L8, MiniPoolStorage.sol#L21-L23, LendingPoolStorage.sol#L4-L8, LendingPoolStorage.sol#L19-L21: The code in this context is unused moreover the use `using ReserveLogic for DataTypes.ReserveData` in `MiniPoolStorage` signalled a copy/paste mistake.

The following patch can be applied:

```
diff --git a/contracts/protocol/core/lendingpool/LendingPoolStorage.sol
↪  b/contracts/protocol/core/lendingpool/LendingPoolStorage.sol
index 7bf3041..ad00f83 100644
--- a/contracts/protocol/core/lendingpool/LendingPoolStorage.sol
+++ b/contracts/protocol/core/lendingpool/LendingPoolStorage.sol
@@ -1,11 +1,6 @@
 // SPDX-License-Identifier: agpl-3.0
 pragma solidity 0.8.23;

-import {UserConfiguration} from
-    "../../../../contracts/protocol/libraries/configuration/UserConfiguration.sol";
-import {ReserveConfiguration} from
-    "../../../../contracts/protocol/libraries/configuration/ReserveConfiguration.sol";
-import {ReserveLogic} from "../../../../contracts/protocol/core/lendingpool/logic/ReserveLogic.sol";
 import {ILendingPoolAddressesProvider} from
     "../../../../contracts/interfaces/ILendingPoolAddressesProvider.sol";
 import {DataTypes} from "../../../../contracts/protocol/libraries/types/DataTypes.sol";
@@ -16,10 +11,6 @@ import {DataTypes} from "../../../../contracts/protocol/libraries/types/DataType
  * @dev Contract containing storage variables for the LendingPool contract.
  */
 contract LendingPoolStorage {
-    using ReserveLogic for DataTypes.ReserveData;
-    using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
-    using UserConfiguration for DataTypes.UserConfigurationMap;
-
     /// @dev The addresses provider contract reference.
     ILendingPoolAddressesProvider internal _addressesProvider;

diff --git a/contracts/protocol/core/minipool/MiniPoolStorage.sol
↪  b/contracts/protocol/core/minipool/MiniPoolStorage.sol
```

```
index 829a4ad..c861dc4 100644
--- a/contracts/protocol/core/minipool/MiniPoolStorage.sol
+++ b/contracts/protocol/core/minipool/MiniPoolStorage.sol
@@ -1,11 +1,6 @@
 // SPDX-License-Identifier: BUSL-1.1
 pragma solidity 0.8.23;

-import {UserConfiguration} from
-    "../../../../contracts/protocol/libraries/configuration/UserConfiguration.sol";
-import {ReserveConfiguration} from
-    "../../../../contracts/protocol/libraries/configuration/ReserveConfiguration.sol";
-import {ReserveLogic} from "../../../../contracts/protocol/core/lendingpool/logic/ReserveLogic.sol";
 import {IMiniPoolAddressesProvider} from
     "../../../../contracts/interfaces/IMiniPoolAddressesProvider.sol";
 import {ILendingPool} from "../../../../contracts/interfaces/ILendingPool.sol";
@@ -18,10 +13,6 @@ import {DataTypes} from "../../../../contracts/protocol/libraries/types/DataType
  * @author Cod3x
  */
 contract MiniPoolStorage {
-    using ReserveLogic for DataTypes.ReserveData;
-    using ReserveConfiguration for DataTypes.ReserveConfigurationMap;
-    using UserConfiguration for DataTypes.UserConfigurationMap;
-
     /// @dev The addresses provider contract managing this MiniPool's addresses.
     IMiniPoolAddressesProvider internal _addressesProvider;
```

- MiniPoolAddressProvider.sol#L158-L160: `getAToken6909` has the same logic as `getMiniPoolToAERC6909` which is used more. `getAToken6909` is only used in test cases and also declared in the interface. Perhaps can be removed.
- ATokenERC6909.sol#L64-L65: `_totalTokens` is not used.
- ATokenERC6909.sol#L68-L69: `_totalTrancheTokens` is also really not used except being incremented and there is not even a `public` getter function for this value. If a newer implementation is not going to use this parameter it can be removed or at least provide a public getter function for it.
- ATokenERC6909.sol#L361: `handleRepayment` is a no-op, unless we add a logic here to track the underlying sent to this contract per token id.
- WadRayMath.sol#L127-L128, WadRayMath.sol#L139-L140: The error part of the `require` statement is unreachable, since if there was an overflow it would already revert on the line before. Note that the original `Aave 2.0` was using the `solc` pragma version `0.6.12`, but here we are using `0.8.23`.
- MiniPoolWithdrawLogic.sol#L233: `internalWithdraw`'s return parameter is not used.
- ATokenERC6909.sol#L539, ATokenERC6909.sol#L577: Redundant check. `_setUnderlyingAsset` has this check.
- ATokenERC6909.sol#L286-L289: Redundant check already performed in `_decreaseBorrowAllowance`.
- MiniPoolFlashLoanLogic.sol#L261-L268, FlashLoanLogic.sol#L273-L280: Emitting this event here seems redundant, since it is also emitted at the end of the `for` loop where `_handleFlashLoanRepayment` has been called (same for the mini pool).

Cod3x: Fixed in commit 5de91bf5.

Spearbit: Fix verified.

5.5.2 `AddressesProvider` ids can be hashed then stored

Severity: Informational

Context: LendingPoolAddressesProvider.sol#L20-L26, MiniPoolAddressProvider.sol#L68-L72

Description/Recommendation: It might be best to store hash of the values in this context in case longer names need to be assigned but still easy to derive the `id`:

```
bytes32 private constant X = keccak256("X");
```

Cod3x: Fixed in commit eb871663.

Spearbit: Fix verified.

5.5.3 Use `abi.encodeCall` instead of the other `abi.encode...` variants

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description/Recommendation:

- LendingPoolAddressesProvider.sol#L173, MiniPoolAddressProvider.sol#L467: Define a new interface like `IAddressProviderUpdatable` and make sure `newAddress` is of this type. Then in the `abi` encoding instead use the following:

  ```
  bytes memory params = abi.encodeCall(IAddressProviderUpdatable.initialize, (address(this)));
  ```

  This will make sure potential future mistakes will be avoided.

- MiniPoolAddressProvider.sol#L278, MiniPoolAddressProvider.sol#L290, MiniPoolAddressProvider.sol#L318

Cod3x: Fixed in commit ff14d5c9.

Spearbit: Fix verified.

5.5.4 The contracts that inherit from `VersionedInitializable` should be made uninitialisable to avoid potential mistakes

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description/Recommendation:

- LendingPool.sol#L51, LendingPoolConfigurator.sol#L31, MiniPool.sol#L51, MiniPoolConfigurator.sol#L26, VersionedInitializable.sol#L16, AToken.sol#L25, VariableDebtToken.sol#L21, ATokenERC6909.sol#L30: The contracts that inherit from `VersionedInitializable` should be made uninitialisable to avoid potential mistakes. Only the proxy contracts using these contracts as implementations should be able to initialise. In the constructors you would need to set the `lastInitializedRevision` to `type(uint256).max`.

Cod3x: Fixed in commit 0983f1ea.

Spearbit: Fix verified.

5.5.5 Formatting, typos, Comments, minimal suggestions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description/Recommendation:

- AToken.sol#L77-L90: Public storage parameters are prefixed with underscore _.
- Token.sol#L90-L99: Inconsistent empty line spacings for the modifier NatSpec.
- AToken.sol#L507: `CALLER_NOT_WRAPPER` should be defined in `Errors.sol`.
- VariableDebtToken.sol#L199-L275: The lines of the form `X;` where `X` is just a variable can be removed.

- ATokenERC6909.sol#L114-L116: `name`, `symbol` and `decimals` shadow storage parameter names.
- ATokenERC6909.sol#L131: Typo: `Rebasin` → `Rebasing`.
- Rewarder.sol#L54, Rewarder6909.sol#L54, RewardsController.sol#L154, RewardsController.sol#L394, Rewarder6909.sol#L54, RewardsController6909.sol#L310, BasePiReserveRateStrategy.sol#L275, BasePiReserveRateStrategy.sol#L291, BasePiReserveRateStrategy.sol#L306, BasePiReserveRateStrategy.sol#L328, BasePiReserveRateStrategy.sol#L342: Internal functions without an underscore `_` prefix.
- RewardsDistributor.sol#L54: Add a `getIsRewardEnabled()` function to retrieve the value of the `_isRewardEnabled` internal variable.
- WadRayMath.sol#L156-L158: No custom errors are thrown compared to `rayMul`.
- InitializableImmutableAdminUpgradeabilityProxy.sol#L25: The `admin` input shadows a public function name `admin()` from `BaseImmutableAdminUpgradeabilityProxy`.
- Oracle.sol#L39-L43: The `notice` NatSpec seems misplaced.
- BorrowLogic.sol#L339: It would be best to capitalise `repayParams`.
- ReserveLogic.sol#L198: `//updated` → `// updated`. Space is missing.
- MiniPoolLiquidationLogic.sol#L191-L209: Here we are burning the `vars.debtID` in `vars.collateralAtoken` which is:

```
vars.collateralAtoken = IAERC6909(collateralReserve.aTokenAddress);
```

```
collateralReserve.aTokenAddress
    debtReserve.aTokenAddress
```

  These two `aTokenAddress` addresses are the same with the current implementation since the `MiniPoolAddressesProvider` can only update the implementation logic of the `aTokenAddress`. This proxy which is linked to the mini pool is linked to all the reserves in the mini pool and they all have the same address. It is important to at least leave a comment regarding this to avoid potential mistakes.
- LendingPoolConfigurator.sol#L297-L303, MiniPoolConfigurator.sol#L312-L318: This also has another consequence, whenever liquidation happens the health factor improves or gets worse depending on whether:
  1. Health factor gets worse:

$$HF < l_{th} l_b \leq 1 \Rightarrow HF' < HF < 1$$

     In this case after liquidation the position stays unhealthy and so the liquidator can keep liquidating the same collateral/debt reserve combo till almost all user collaterals from that specific reserve are liquidated.
  2. Health factor improves:

$$1 > HF \geq l_{th} l_b \Rightarrow HF' \geq HF.$$

     In this case an unhealthy position could potentially become healthy after liquidation.

parameter	description
HF	health factor normalised before liquidation
HF'	health factor normalised after liquidation
l_{th}	`liquidationThreshold` normalised
l_b	`liquidationBonus` normalised

- BorrowLogic.sol#L292-L297: Unlike the regular lending pool borrow flow where minting the debt token checks whether we should flag in the user config whether the asset in the question is being borrowed in the `executeMiniPoolBorrow` flow this return value is not checked for the `params.miniPoolAddress` and thus when querying the user config for this `params.miniPoolAddress` we would not see any assets borrowed. It might be best to leave a comment here and mention that this would always cause the health factor check to be passed (unbacked borrows by mini pools in the system).
- FlowLimiter.sol#L24-L27, MiniPoolPiReserveInterestRateStrategy.sol#L46: The storage parameters in this context can be `immutable`.
- LiquidationLogic.sol#L322-L323: On the mini pool side these stack parameters in the context have been explicitly set as `0`:

```
uint256 collateralAmount = 0;
uint256 debtAmountNeeded = 0;
```

- BasePiReserveRateStrategy.sol#L328-L349: It would be best to move these functions to `ReserveConfiguration` library. Their counterparts with `storage` parameter inputs are already defined in that library.
- Oracle.sol#L31: Getter function for `_assetToTimeout` is missing.
- ATokenERC6909.sol#L175: Reuse `underlyingAsset` instead of `_underlyingAssetAddresses[id]`.
- BasePiReserveRateStrategy.sol#L121-L125: `_getLendingPool` has a confusing naming. Also the lending pool and main pool considered the same in the NatSpec which contradicts the naming convention in the folders/files.
- MiniPool.sol#L105: Kind of related to the previous point where `Errors.LP_CALLER_NOT_LENDING_POOL_CONFIGURATOR` is used in the context of the mini pool. The error mentions lending pool configurator although this check is for the mini pool configurator.
- MiniPool.sol#L30-L34: Mixed use of import path styles applies to other files as well.
- ILendingPool.sol#L12, IMiniPool.sol#L12: `initialize` is missing from the interface.
- LendingPoolConfigurator.sol#L211, LendingPoolConfigurator.sol#L179: Safe cast `decimals` to `uint8`.
- LendingPoolConfigurator.sol#L178, LendingPoolConfigurator.sol#L210: Cast `input.incentivesController` to `IRewarder`.
- MiniPool.sol#L737-L744: Leave a note that multiple reserves might have the same `aTokenAddress` proxy and thus change the `rewarder` for one changes it for all those reserves.
- MiniPool.sol#L141: `deposit` is a `public` function for the mini pool but for the lending pool it is an `external` function.
- ReserveLogic.sol#L306-L307: Outdated comment since in contrary to Aave v2 there are no `StableDebtTokens` in this protocol.
- ReserveLogic.sol#L141, MiniPoolReserveLogic.sol#L140: Simplify to:

```
uint256 amountToLiquidityRatio = amount.rayDiv(totalLiquidity);
```

  Note that as long as the denominator and nominator of the `rayDiv` have the same unit the final result will be in `ray`.
- FlashLoanLogic.sol#L175: For the lending pool logic the `reserves[vars.currentAsset][vars.currentType]` counterpart has been cached in a `reserve` storage parameter.
- MiniPoolFlashLoanLogic.sol#L147-L152: `DataTypes.MiniPoolReserveData storage reserve` can be declared in the `if` block's body instead just before `_handleFlashLoanRepayment`.
- FlashLoanLogic.sol#L129-L131, MiniPoolFlashLoanLogic.sol#L112-L114: Validating that the `flashLoanParams.modes` has the same length as the other arrays is missing.

- FlashLoanLogic.sol#L239, MiniPoolFlashLoanLogic.sol#L223: Out of range value for casting from `uint256` to `DataTypes.InterestRateMode` which be checked here:

```
DataTypes.InterestRateMode(modes[i])
```

  Perhaps a comment can be added.

- MiniPool.sol#L222: Comment should be:

```
// amount - availableLiquidity converted to asset
```

  `+` → `-`.

- MiniPoolBorrowLogic.sol#L326: We could set `receiverOfUnderlying` to `address(0)` since it is not going to be used. It would be cheaper too.

- ValidationLogic.sol#L306: Does not check that `reserveType.length` is also the same as the other array lengths.

- BorrowLogic.sol#L284-L288: Best to do this check in `flowLimiter` so that one would only call this contract once here instead of twice.

```
flowLimiter.revertIfFlowLimitReached(params.asset, params.miniPoolAddress, params.amount);

// rest of the logic does not need to be in an `else` block.
```

- MiniPoolDepositLogic.sol#L104: `reserve.aTokenAddress` is already cached in `aToken` and can be reused.

- AToken.sol#L576-L584: Note that if either one of these `if` / `else` blocks is entered `_farmingBal` ends up being `finalFarmingAmount`. But if non-entered it would stay the same within a proportional neighbourhood of `finalFarmingAmount` (bounded proportional error/tolerance).

  Possibly cheaper (but current version is more readable):

```
if (pctOfFinalBal > _farmingPct && pctOfFinalBal - _farmingPct > _farmingPctDrift) {
    // We will end up overallocated, withdraw some.
    toWithdraw = currentAllocated - finalFarmingAmount;
    _farmingBal = finalFarmingAmount;
} else if (pctOfFinalBal < _farmingPct && _farmingPct - pctOfFinalBal > _farmingPctDrift) {
    // We will end up underallocated, deposit more.
    toDeposit = finalFarmingAmount - currentAllocated;
    _farmingBal = finalFarmingAmount;
}
```

  Just a note that the assignment `_farmingBal = finalFarmingAmount` cannot be refactored out of the `if` / `else` blocks since there is a possibility that non of the conditions are true and `_farmingBal` would need to stay the same.

- MiniPoolConfigurator.sol#L112: Implementation is slightly different from the lending pool configurator counter part, for example the `FlashloanPremiumTotalUpdated` event is not emitted here.

- BorrowLogic.sol#L441: In the `repayWithAtokens` flow where the `params.onBehalfOf` is the mini pool the reserve is never marked as being borrowed in the first place.

- MiniPoolDefaultReserveInterestRate.sol#L147, MiniPoolDefaultReserveInterestRate.sol#L151, MiniPoolPiReserveInterestRateStrategy.sol#L106, MiniPoolPiReserveInterestRateStrategy.sol#L110: `IERC20(asset).balanceOf(aToken)` can be manipulated by donations which affect the utilisation rate (make it smaller), it might make sense to track the assets managed by `aToken` in the context of the mini pools just like the `aTokens` in the context of lending pool.

- RewardsController.sol#L135-L136: Rename `assetID` to something more specific, like `miniPoolREserveATokenID` or just `aTokenID`.

- Oracle.sol#L162: `// If "asset" is an aToken then convert the price from asset to share. → // If "asset" is an aToken then convert the price from share to asset.`
- ATokenERC6909.sol#L60:

```
- IMiniPoolRewarder private INCENTIVES_CONTROLLER;
+ IMiniPoolRewarder private _incentivesController;
```

- ATokenERC6909.sol#L284: use `isDebtToken(id)`.

Cod3x: Fixed in commit 70909f99.

Spearbit: Fix verified.

5.5.6 Some contracts inherit from `Context` but instead of `_msgSender` use `msg.sender`

Severity: Informational

Context: AToken.sol#L478, AToken.sol#L507

Description: The contracts in this context inherit from `Context` but instead of `_msgSender` use `msg.sender`.

Recommendation: Either remove the inheritance from `Context` from these contracts or make sure to use `_msgSender` whenever `msg.sender` is used.

Cod3x: Fixed in commit 2c193bd4.

Spearbit: Fix verified.

5.5.7 The NatSpec comment for `wadDiv` and `rayDiv` are not entirely accurate

Severity: Informational

Context: WadRayMath.sol#L105-L118, WadRayMath.sol#L60-L73

Description: The NatSpec in this context mentions that:

```
/**
 * @notice Divides two X numbers, rounding half up to the nearest X. <----
 ...
 */
function xDiv(uint256 a, uint256 b) internal pure returns (uint256) {
    // ...
```

The portion that gets rounded up depends on `b` it might not be always from half up (but very close when `b` is odd).

If $b = 2k + 1$ then the function:

$$f(x) = \left\lfloor \frac{x + k}{2k + 1} \right\rfloor = \left\lfloor \frac{x}{b} + \frac{1}{2 + 1/k} \right\rfloor$$

only rounds up values when the fractional part of $\frac{x}{b}$ is in the interval $[1 - \frac{1}{2+1/k}, 1)$. The comment is correct for even values of b.

Recommendation: It might not be as important but perhaps it might be best to be documented.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.5.8 `input.underlyingAssetDecimals` can be derived from `input.underlyingAsset` to avoid potential mistakes

Severity: Informational

Context: LendingPoolConfigurator.sol#L108, LendingPoolConfigurator.sol#L123, LendingPoolConfigurator.sol#L142, MiniPoolConfigurator.sol#L210, MiniPoolConfigurator.sol#L222

Description/Recommendation: `input.underlyingAssetDecimals` can be derived from `input.underlyingAsset` to avoid potential mistakes.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.5.9 Addresses provided to `FlowLimiter` constructor can be derived from only one address

Severity: Informational

Context: FlowLimiter.sol#L37-L40

Description/Recommendation: Instead of providing both of these addresses below one can provide just one and derive the other or one can provide the lending pool address provider and derive both of these addresses:

```
constructor(IMiniPoolAddressesProvider miniPoolAddressesProvider, ILendingPool lendingPool) {
    _lendingPool = lendingPool;
    _miniPoolAddressesProvider = miniPoolAddressesProvider;
}
```

Cod3x: Fixed in commit 21b2fa72.

Spearbit: Fix verified.

5.5.10 Storage collisions and different styles of picking storage slots

Severity: Informational

Context: ATokenERC6909.sol#L30

Description/Recommendation: Note that the way `_ERC6909_MASTER_SLOT_SEED` is picked and used in calculating the storage slots in `solday`'s `ERC6909` implementation storage collision for the `mapping` types in the `ATokenERC6909` contract would not be possible. But perhaps one should leave a note for potential future mistakes.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.5.11 `ATokenERC6909.{mint, burn}` can be simplified

Severity: Informational

Context: ATokenERC6909.sol#L284-L301, ATokenERC6909.sol#L324-L333

Description/Recommendation: `ATokenERC6909.mint` can be simplified to:

```
if (isDebtToken(id) && onBehalfOf != user) {
    _decreaseBorrowAllowance(onBehalfOf, user, id, amount);
}

uint256 previousBalance = super.balanceOf(onBehalfOf, id);
uint256 amountScaled = amount.rayDiv(index);
require(amountScaled != 0, Errors.CT_INVALID_MINT_AMOUNT);
_mint(onBehalfOf, id, amountScaled);
```

`ATokenERC6909.burn` can be simplified to:

```
uint256 amountScaled = amount.rayDiv(index);
require(amountScaled != 0, Errors.CT_INVALID_BURN_AMOUNT);
_burn(user, id, amountScaled);

if (isAToken(id)) {
    transferUnderlyingTo(receiverOfUnderlying, id, amount, unwrap);
}
```

Cod3x: Fixed in commit 21b2fa72.

Spearbit: Fix verified.

5.5.12 The code to derive `minLiquidityRate` can be simplified

Severity: Informational

Context: MiniPoolDefaultReserveInterestRate.sol#L231-L241, MiniPoolPiReserveInterestRateStrategy.sol#L202-L212

Description/Recommendation: The code to derive `minLiquidityRate` can be simplified:

```
uint256 commonTerm = (r.currentLiquidityRate * DELTA_TIME_MARGIN / SECONDS_PER_YEAR) + WadRayMath.ray();
uint256 minLiquidityRate = (
    MathUtils.calculateCompoundedInterest(
        r.currentVariableBorrowRate, uint40(block.timestamp - DELTA_TIME_MARGIN)
    ) - commonTerm
).rayDiv(commonTerm * DELTA_TIME_MARGIN / SECONDS_PER_YEAR);
```

Cod3x: Fixed in commit 21b2fa72.

Spearbit: Fix verified.

5.5.13 `setPause` can be called with the same input as the previous value

Severity: Informational

Context: MiniPool.sol#L695-L702, LendingPool.sol#L676-L683

Description:

1. The `setPause` endpoint allows the pool configurator to pause a paused pool and to unpause an unpaused pool. In terms of emitting events, it might be confusing for off-chain analysis.
2. It might be cheaper to reuse `val` instead of `_paused` for the conditional statement.

Recommendation:

1. It might be best to compare the old and new value and if they are the same revert.
2. Reuse `val` for the conditional statement in the `if` block.

Cod3x: Fixed in commit 21b2fa72.

Spearbit: Fix verified.

5.5.14 `BorrowLogic::calculateUserAccountDataVolatile` could use `GenericLogic`

Severity: Informational

Context: BorrowLogic.sol#L99-L104, MiniPoolBorrowLogic.sol#L101-L106

Description: The function `calculateUserAccountDataVolatile` could reuse the logic in `GenericLogic.calculateUserAccountData`.

Recommendation: Refactor `calculateUserAccountDataVolatile` as follows in `MiniPoolBorrowLogic` and `BorrowLogic`:

```
function calculateUserAccountDataVolatile(
    CalculateUserAccountDataVolatileParams memory params,
    mapping(address => mapping(bool => DataTypes.ReserveData))
        storage reserves,
    DataTypes.UserConfigurationMap memory userConfig,
    mapping(uint256 => DataTypes.ReserveReference) storage reservesList
) external view returns (uint256, uint256, uint256, uint256, uint256) {
    return GenericLogic.calculateUserAccountData(
        params.user,
        reserves,
        userConfig,
        reservesList,
        params.reservesCount,
        params.oracle
    );
}
```

Cod3x: Fixed in commit 21b2fa72.

Spearbit: Fix verified.

5.5.15 Useless `transferFunction` negative checks

Severity: Informational

Context: BasePiReserveRateStrategy.sol#L108-L110, BasePiReserveRateStrategy.sol#L168

Description/Recommendation: In `InterestRateStrategy` implementations, checking that `transferFunction` is not negative are useless, since `transferFunction` returns a `uint256`.

Cod3x: Fixed in commit 21b2fa72.

Spearbit: Fix verified.

5.5.16 Minipool market ERC6909 address handling could be improved

Severity: Informational

Context: DataTypes.sol#L51-L63

Description: Currently minipool reserves data holds the address `aTokenAddress` which is the address of the `ERC6909` contract which handles all of the liquidity and debt tokens for the minipool.

```
struct MiniPoolReserveData {
    ReserveConfigurationMap configuration;
    uint128 liquidityIndex;
    uint128 variableBorrowIndex;
    uint128 currentLiquidityRate;
    uint128 currentVariableBorrowRate;
    uint40 lastUpdateTimestamp;
    address aTokenAddress;
    uint256 aTokenID;
    uint256 variableDebtTokenID;
    address interestRateStrategyAddress;
    uint8 id;
}
```

There are two drawbacks with this approach:

1. This name is misleading, because not only aTokens are handled but debt tokens. The name used in the Rewarder `market6909` is better suited.
2. Having the address available in the reserve struct is misleading, because it may imply that reserves can have different markets, when only one is ever created for a given minipool.

This is especially confusing in contexts where multiple reserves are handled such as liquidations.

Recommendation: Please consider renaming `aTokenAddress` in minipool related contracts to `marketErc6909`, and keep the address available from the minipool configuration or address provider.

Cod3x: Fixed in commit 6a7c4703.

Spearbit: Fix verified.

5.5.17 Lending pool's borrowed reserves by the mini pools are not marked in user configs

Severity: Informational

Context: MiniPool.sol#L219-L231

Description: The `reserve` in the lending pool will be marked as collateral for the mini pool in the lending pool's user config but the `miniPoolBorrow` does not mark this reserve as a borrowed reserve for a the mini pool.

Recommendation: It would be best to mark the borrowed reserve in the lending pool by the mini pool in its user config. But it is important to skip the health checks for the mini pool during the borrowing and liquidation. And thus some customised logic also needs to be added to the health check of mini pools in lending pools.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.5.18 Interacting with a reserve can be locked If a mini pool incures bad debt for the reserve borrowed from the lending pool

Severity: Informational

Context: MiniPoolDefaultReserveInterestRate.sol#L233-L234, MiniPoolPiReserveInterestRateStrategy.sol#L212-L213

Description: If a mini pool incures bad debt (subtraction below reverting due to arithmetic underflow) for the reserve borrowed from the lending pool interacting with that reserve in mini pool can be locked:

```
MathUtils.calculateCompoundedInterest(
    r.currentVariableBorrowRate, uint40(block.timestamp - DELTA_TIME_MARGIN)
) - r.currentLiquidityRate * DELTA_TIME_MARGIN / SECONDS_PER_YEAR - WadRayMath.ray()
```

In normal conditions this should not happen since `r.currentLiquidityRate` should be a fraction of `r.currentVariableBorrowRate` and the first term above is roughly:

```
WadRayMath.ray() + r.currentVariableBorrowRate * DELTA_TIME_MARGIN / SECONDS_PER_YEAR
```

and so the subtraction should be roughly:

```
(r.currentVariableBorrowRate - r.currentLiquidityRate) * // ...
```

Recommendation: This would need further analysis and to unlock the reserve, the admins can always deploy a new `IMiniPoolReserveInterestRateStrategy` and swap the old one for the mini pool with the newly deployed one.

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.5.19 Unused entry `rewardedToken` in `RewardForwarder.claimedRewards`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: RewardForwarder.sol uses a triple mapping to handle balances of rewards received:

- RewardForwarder.sol#L30-L35:

```
/**
 * @dev Mapping to track claimed rewards per claimee.
 * Maps `claimee` => `rewardedToken` => `rewardTokensIndex` => `amount`.
 */
mapping(address => mapping(address => mapping(uint256 => uint256))) public claimedRewards;
```

However the middle entry `rewardedToken`, is never used, and could be removed to simplify reward forwarding logic (no need to call `forwardRewards` for every `rewardedToken`, only for every `rewardTokenIndex`):

```
/**
 * @notice Forwards previously claimed rewards to the designated forwarder.
 * @dev Requires claimed rewards to exist and a forwarder to be set.
 * @param claimee The address of the claimee.
 * @param rewardedToken The address of the rewarded token.
 * @param rewardTokenIndex The index of the reward token in the `rewardTokens` array.
 */
function forwardRewards(address claimee, address rewardedToken, uint256 rewardTokenIndex) public
↪  {
    address rewardToken = rewardTokens[rewardTokenIndex];
    uint256 amount = claimedRewards[claimee][rewardedToken][rewardTokenIndex];
    require(amount != 0, "No rewards to forward");
    claimedRewards[claimee][rewardedToken][rewardTokenIndex] = 0;
    address forwarder = forwarders[claimee][rewardTokenIndex];
    require(forwarder != address(0), "No forwarder set");
    IERC20(rewardToken).transfer(forwarder, amount);
}
```

Recommendation: Please consider removing `rewardedToken` from the `claimedRewards` mapping.

Cod3x: Fixed in commit e1881495.

Spearbit: Fix verified.

5.5.20 Minipool upgradability could be simplified using beacon proxy pattern

Severity: Informational

Context: MiniPoolAddressProvider.sol#L312-L316

Description: In current implementation, minipools are individually deployed by creating a proxy and providing an implementation address:

- MiniPoolAddressProvider.sol#L305-L316:

```
/**
 * @dev Deploys a new mini pool with associated contracts.
 * @param miniPoolImpl The mini pool implementation address.
 * @param aTokenImpl The aToken implementation address.
 * @param poolAdmin The admin address for the new pool.
 * @return The ID of the newly created mini pool.
 */
function deployMiniPool(address miniPoolImpl, address aTokenImpl, address poolAdmin)
    external
    onlyOwner
    returns (uint256)
{
    // ...
}
```

These minipools implementations are then updated manually:

- MiniPoolAddressProvider.sol#L276:

```
function setMiniPoolImpl(address impl, uint256 miniPoolId) external onlyOwner {
    // ...
}
```

In this case, if the logic of minipools must be updated, a call must be made to all minipools. Since it makes sense to keep minipool implementations consistent, this setup could use the beacon proxy pattern.

Recommendation: Change the minipool proxy type to be a beacon type proxy, the beacon could still be provided during deployment in case we should be able to create multiple categories of minipools with different implementations:

- MiniPoolAddressProvider.sol#L305-L316:

```
/**
 * @dev Deploys a new mini pool with associated contracts.
 * @param minipoolBeacon The mini pool beacon address.
 * @param aTokenImpl The aToken implementation address.
 * @param poolAdmin The admin address for the new pool.
 * @return The ID of the newly created mini pool.
 */
function deployMiniPool(address minipoolBeacon, address aTokenImpl, address poolAdmin)
    external
    onlyOwner
    returns (uint256)
{
    // ...
}
```

Cod3x: Fixed in commit 45e732fe.

Spearbit: Different route is taken by the client to just add helper utility functions, the mechanics stay the same as each proxy contract's implementation would need to be updated individually.

5.5.21 Redundant `atokenAddress` parameter during `miniPoolBorrow` call

Severity: Informational

Context: MiniPool.sol#L223

Description: Fetching the aToken address from non-rebasing wrapper and passing to the call of `miniPoolBorrow` is redundant:

- MiniPool.sol#L219-L224:

```
  ILendingPool(vars.LendingPool).miniPoolBorrow(
      underlying,
      true,
      ATokenNonRebasing(asset).convertToAssets(amount - vars.availableLiquidity), // amount +
      ↪  availableLiquidity converted to asset
      ATokenNonRebasing(asset).ATOKEN_ADDRESS()
  );
```

Since we are making the call to the main lending pool, it will be able to deduce it from underlying directly.

Recommendation: In MiniPool.sol#L219-L224:

```
  ILendingPool(vars.LendingPool).miniPoolBorrow(
      underlying,
      true,
-     ATokenNonRebasing(asset).convertToAssets(amount - vars.availableLiquidity), // amount +
↪  availableLiquidity converted to asset
-     ATokenNonRebasing(asset).ATOKEN_ADDRESS()
+     ATokenNonRebasing(asset).convertToAssets(amount - vars.availableLiquidity) // amount +
↪  availableLiquidity converted to asset
  );
```

Cod3x: Acknowledged.

Spearbit: Acknowledged.

5.5.22 Concerns about initializing new reserves

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description/Recommendation: During AAVE V2 deployments of new reserves, they initialize each reserve with a small amount of initial supply. Failing to add this initial supply introduces vulnerabilities to the code. Some past hacks, such as the Radiant Hack, occurred due to missing initial liquidity in new reserves.

The fix can be implemented in deployment scripts of new reserves by adding a small amount of liquidity before initialization.

Cod3x: Acknowledged, this is handled in the deployment scripts.

Spearbit: Acknowledged.