Electisec Monolith Review

Review Resources:

- None beyond the code repositories

Auditors:

- Fedebianu
- Invader

Table of Contents

Review Summary

Monolith protocol provides a decentralized platform that allows users to manage and borrow stablecoins in single-collateral markets. The mechanism that sets Monolith aside is that the user can opt into a debt-free mode, where they aren't charged interest on their loan but, in exchange, could get their collateral redeemed by external actors - conceptually similar to how a Liquity trove works. The protocol utilizes this mechanism to gauge the user's risk appetite and govern the interest rate model while also serving as a means to protect the peg.

The contracts of the Monolith protocol repo were reviewed over five days. Two auditors conducted the code review between May 20th and 27th, 2025. The review was limited to the latest commit on the dev branch of the Monolith repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
.
├── Vault.sol
├── Lender.sol
├── InterestModel.sol
├── Factory.sol
└── Coin.sol
```

This review is a code review aimed at identifying potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the code's security

relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

Electisec and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. Electisec and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Monolith and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	The protocol implements operator-based access control for critical functions.
Mathematics	Good	The protocol utilizes fixed-point arithmetic libraries (Solmate's FixedPointMathLib) and a custom interest rate model (InterestModel.sol). While the foundational math libraries are standard, some of the choices done, for instance, the handling of the interest rate model, are complex and could benefit from additional documentation to aid understanding.
Complexity	Good	The system involves several interconnected components (Lender, Vault, Coin, Factory, InterestModel) that manage collateralized debt positions, liquidations, and dynamic interest rates. While the architectural complexity is rated 'Good', the way the protocol handles free debt position and redemption is novel. It will likely have complex interactions with external entities that could be difficult to predict.
Libraries	Good	The protocol uses Solmate libraries for core functionalities like ERC20, ERC4626, and safe transfers.
Decentralization	Good	The protocol features an operator role with significant privileges over certain parameters and functions. However, there are mechanisms for operator transition and a planned immutability deadline.
Code stability	Good	The code was not under active development during the audit; the lens contract was out of scope for the audit, which may indicate a need for further security reviews.
Documentation	Average	NatSpec documentation is present but is sparse in many areas and does not consistently cover all functions, parameters, and return values. Existing comments are sometimes misleading (e.g. in the getCollateralPrice() and getLiquidatableDebt() methods). More comprehensive and accurate documentation, including detailed explanations of complex mechanisms, would be desirable.
Monitoring	Average	The contracts emit events for several key operations. However, some state changes and actions lack events - e.g., certain setters in Factory.sol and under certain circumstances when positions are adjusted.
Testing and Verification	Average	The test coverage is good, and the inclusion of fuzzing tests is positive. Still, there is a lack of more extensive testing exploring different scenarios

Category	Mark	Description
		(or if it exists, it's not part of the repo), which would be recommended considering the novelty of the freeDebt and redemption mechanism.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

1. Critical - `accrueInterest()` is compromised

`lastAccrue` starts at 0, causing `accrueInterest()` to either revert or calculate ~55 years of interest in one transaction, permanently corrupting all calculations and rendering the protocol unusable.

Technical Details

In `accrueInterest()`, there is an incorrect calculation of `timeElapsed` that uses the uninitialized state variable `lastAccrue`. This causes `timeElapsed` to be equal to `block.timestamp` on the first call, which is a very large number. This large `timeElapsed` leads

`calculateInterest()` to always revert until `lastFreeDebtRatioBps > targetFreeDebtRatioEndBps`.

However, when the function finally stops reverting, the system calculates interest for an enormous time period.

Proof of concept

Insert this in `InterestModel.t.sol`:

```
function test_calculateInterest_poc() public view {
    // Setup test parameters
    uint totalPaidDebt = 0;
    uint lastFreeDebtRatioBps = 0; // 0%
    uint timeElapsed = 1_747_920_023; // use actual timestamp
    uint lastRate = 5e15; // 0.5%

    uint expRate = uint(wadLn(2*1e18)) / 7 days;
    uint targetFreeDebtRatioStartBps = 2000; // 20%
    uint targetFreeDebtRatioEndBps = 4000; // 40%

    interestModel.calculateInterest(
        totalPaidDebt,
        lastRate,
        timeElapsed,
        expRate,
        lastFreeDebtRatioBps,
        targetFreeDebtRatioStartBps,
        targetFreeDebtRatioEndBps
    );
}
```

Results:

```
Ran 1 test for test/InterestModel.t.sol:InterestModelTest
[FAIL: panic: division or modulo by zero (0x12)] test_calculateInterest_poc() (gas:
9940)
```

```
Logs:
  growthDecay 0
```

Impact

Critical. `accrueInterest()` is compromised, leading to the failure of the entire system.

Recommendation

Initialize `lastAccrue`.

Developer Response

Fixed in commits 395e6d9. Also initialized `cachedGlobalFeeBps` which was defaulting to 0 and causing tests to fail after initializing `lastAccrue` in commit e622fb8.

Medium Findings

1. Medium - Collateral from `writeOff()` becomes permanently locked in `Lender` contract

`writeOff()` transfers borrower collateral to the `Lender` contract during liquidation; however, there is no mechanism to utilize or recover this collateral, rendering it permanently locked and economically useless.

Technical Details

During liquidation `writeOff()` is automatically called via external call:

```
try this.writeOff(borrower) {} catch {}
```

The function, if `debt > collateralValue * 100`, transfers all remaining collateral to the caller. Since `writeOff()` is called via `this.writeOff()`, the `msg.sender` becomes the `Lender` contract itself. The collateral is transferred to the `Lender` contract, but there are no functions to manage this collateral.

Impact

Medium. Over time, significant amounts of collateral could accumulate in the contract as dead capital.

Recommendation

Consider implementing a mechanism to utilize recovered collateral or transfer it to the original caller.

Developer Response

Fixed in commit 3a84b27.

Low Findings

1. Low - Oracle price fallback is not necessary

Technical Details

In `getCollateralPrice()`, when the oracle fails or returns an invalid price, the function sets `price = 1` as a fallback:

```
price = price == 0 ? 1 : price; // avoid division by zero in consumer functions
```

This fallback value of 1 is not necessary as if the price is zero, any other functions return or revert as `reduceOnly` is set to `true` and `allowLiquidations` is set to `false.`

Impact

Low. The function returns an incorrect price value instead of the actual invalid price.

Recommendation

Remove that line of code.

Developer Response

Acknowledged. We have decided to keep this line as an extra precaution against division by zero.

2. Low - `getDebtOf()` returns inconsistent values after epoch changes

`getDebtOf()` returns incorrect debt values for users when called between an epoch change and the user's next interaction with the protocol, leading to inconsistent state reporting and potential integration issues.

Technical Details

`getDebtOf()` is a view function that calculates user debt based on their current shares and the global debt/shares ratio. However, this function does not account for epoch changes that occur during redemptions. When an epoch change happens:

1. `totalFreeDebtShares` is divided by `1e18` in `redeem()`
2. Individual user shares (`freeDebtShares[account]`) remain unchanged until `updateBorrower()` is called
3. This creates a temporary inconsistency where the debt calculation uses outdated user shares

Impact

Low. This inconsistency can cause:

- **Frontend/UI issues**: Incorrect debt balances displayed to users
- **Integration problems**: External protocols reading incorrect debt values
- **Liquidation miscalculations**: Bots may attempt invalid liquidations based on wrong debt amounts
- **User confusion**: Apparent debt spikes that resolve after user interaction

The issue is temporary and resolves when users interact with the protocol, triggering `updateBorrower()`, but can persist indefinitely for inactive users.

Recommendation

Consider adding `updateBorrower()` to the public `getDebtOf()` function and creating a separate internal function for internal usage:

```
function getDebtOf(address account) public returns (uint) {
    updateBorrower(account);
    return _getDebtOf(account);
}

function _getDebtOf(address account) internal view returns (uint) {
    if(isRedeemable[account]) {
        return totalFreeDebtShares == 0 ? 0 :
freeDebtShares[account].mulDivDown(totalFreeDebt, totalFreeDebtShares);
    } else {
        return totalPaidDebtShares == 0 ? 0 :
paidDebtShares[account].mulDivDown(totalPaidDebt, totalPaidDebtShares);
    }
}
```

Developer Response

Acknowledged. `getDebtOf` is primarily meant to be used internally. External consumers are expected to use an out-of-scope `Lens` contract in order to access up-to-date info about `Lender` state. Therefore, no changes will be made to the `Lender` source.

3. Low - The entire `totalFreeDebt` can't be redeemed due to division by zero protection in `redeem()`

`redeem()` contains a division-by-zero check that prevents the complete redemption of all free debt.

Technical Details

`redeem()` includes an intentional division by zero check at the end. This check performs `totalFreeDebtShares / totalFreeDebt`, which will revert with division by zero if `totalFreeDebt` becomes exactly 0. This means complete redemption is impossible as users cannot redeem the very last wei of `totalFreeDebt`.

Impact

Low. The protocol cannot achieve a completely clean state with zero free debt. The last redeemer cannot complete a full redemption and may receive unexpected transaction failures.

Recommendation

Consider adding special handling for complete redemption in `redeem()`:

```
if(totalFreeDebt == 0) {
    epoch++;
    totalFreeDebtShares = 0;
    emit NewEpoch(epoch);
} else if(totalFreeDebtShares / totalFreeDebt > 1e18) {
    epoch++;
    totalFreeDebtShares /= 1e18;
    emit NewEpoch(epoch);
}
```

Handle this case also in `updateBorrower()`:

```
if(epoch > _borrowerEpoch) {
    // reduce the borrower's debt to match shares of the next epoch
    borrowerDebtShares = totalFreeDebt == 0 ? 0 : borrowerDebtShares / 1e18;
    // if the division above rounds down to 0, we skip the redemption
    if(borrowerDebtShares > 0) {
        // Add additional redeemedCollateral
        // in this case, the entire epoch's index is equal to our delta
(epochRedeemedCollateral[_borrowerEpoch + 1] - 0)
        redeemedCollateral += epochRedeemedCollateral[_borrowerEpoch +
1].mulDivUp(borrowerDebtShares, 1e18);
    }
    // update the borrower's debt shares. May be 0 or positive.
    freeDebtShares[borrower] = borrowerDebtShares;
}
```

Developer Response

Acknowledged. This is intended behavior in order to reduce the added code complexity required to handle division by 0 in this edge case. Redeeming is not considered a time-sensitive action (unlike liquidations and write-offs) and therefore there is no need for a change in this case. Users will be required to leave at least 1 wei in `totalFreeDebt` after they redeem.

4. Low - Missing maximum interest rate cap allows unbounded exponential growth

Technical Details

In `calculateInterest()`, when `_lastFreeDebtRatioBps < _targetFreeDebtRatioStartBps`, the interest rate grows exponentially without any upper bound:

```
if (_lastFreeDebtRatioBps < _targetFreeDebtRatioStartBps) {
    uint growthDecay = uint(wadExp(int(_expRate * _timeElapsed)));
    currBorrowRate = _lastRate * growthDecay / 1e18;
    // @audit no max rate check here
}
```

The exponential growth formula `_lastRate * wadExp(_expRate * _timeElapsed) / 1e18` can theoretically reach extremely high values, even overflow, if the free debt ratio stays below target for extended periods.

Impact

Low. Borrowers could face impossibly high interest rates.

Recommendation

Implement a maximum interest rate cap:

```
if (_lastFreeDebtRatioBps < _targetFreeDebtRatioStartBps) {
    currBorrowRate = _lastRate * growthDecay / 1e18;
    if (currBorrowRate > MAX_RATE) {
        currBorrowRate = MAX_RATE;
    }
    interest = _totalPaidDebt * (currBorrowRate - _lastRate) / _expRate / 365 days;
}
```

Developer Response

Acknowledged. After internal discussion, we believe that a very high interest is desirable in this scenario. Borrowers will have the option of either repaying their positions OR switch to the free debt redeemable option until the system reduces the interest rate back down to an affordable

level. Adding a maximum interest rate is also not as simple as suggested in the recommendation. We would have to calculate the interest rate integral up to the max rate which will add some complexity and more room for error. The benefits of this addition will only impact few Monolith stablecoins that may reach this state, if any. In this cases, alternatives exist for impacted users and therefore a fix is not needed.

Informational Findings

1. Informational - Emit `PositionAdjusted` event for all position adjustments

Technical Details

In `adjust()`, the `PositionAdjusted` event is not emitted in several valid cases:

1. When `collateralDelta >= 0 && debtDelta <= 0`:

```
// Skip remaining invariants if caller does not reduce collateral AND does not
increase debt
if(collateralDelta >= 0 && debtDelta <= 0) return;
```

2. When `debtBalance == 0`:

```
// Skip solvency checks if debt is zero
if(debtBalance == 0) return;
```

Impact

Informational.

Recommendation

Emit the `PositionAdjusted` event before each early return to ensure all valid position adjustments are properly tracked.

Developer Response

Fixed in commit bc72de3.

2. Informational - Optimize `liquidate()` function implementation

`liquidate()` has a suboptimal implementation for handling the `repayAmount` parameter, which could lead to unnecessary transaction failures and a less user-friendly experience for liquidators.

Technical Details

The current implementation of the `liquidate()` function includes the following logic for handling the `repayAmount`:

```
if(repayAmount == type(uint256).max) {
    require(liquidatableDebt > 0, "no liquidatable debt");
    repayAmount = liquidatableDebt;
} else {
    require(liquidatableDebt >= repayAmount, "insufficient liquidatable debt");
}
```

This logic requires that `liquidatableDebt` be greater than or equal to `repayAmount`, which can lead to transaction failures if the liquidator specifies an amount greater than the available liquidatable debt.

Impact

Informational.

Recommendation

Consider the following changes:

```
require(liquidatableDebt > 0, "no liquidatable debt");
if(repayAmount > liquidatableDebt) {
    repayAmount = liquidatableDebt;
}
```

Developer Response

Fixed in commit eb902fb.

3. Informational - Misleading comment in `getLiquidatableDebt()`

Technical Details

In `getLiquidatableDebt()` there is a misleading comment that suggests liquidating only the debt above borrowing power:

```
// liquidate only the amount of debt that is above the borrowing power
```

The comment states, "liquidate only the amount of debt that is above the borrowing power", but the actual implementation liquidates 25% of the total debt, not just the excess above borrowing power.

Impact

Informational.

Recommendation

Update the comment to reflect the implementation accurately:

```
// liquidate 25% of the total debt
liquidatableDebt = debt / 4;
```

Developer Response

Fixed in commit 3dba660.

4. Informational - Misleading NatSpec comment in `getCollateralPrice()`

`getCollateralPrice()` contains incorrect documentation that misleads developers about the returned price format, and the `getFeedPrice()` function has unnecessary external visibility.

Technical Details

In `getCollateralPrice()`, the documentation incorrectly states:

```
/// @return price The price in USD with 18 decimals
```

However, the actual implementation in `getFeedPrice()` returns a price with `(36 - tokenDecimals)` decimals.

Incorrect documentation can lead to integration errors, where external developers expect 18-decimal prices but receive prices with different decimal precision.

Impact

Informational.

Recommendation

Correct the NatSpec:

```
/// @return price The price in USD normalized to (36 - collateral decimals) decimals
for consistent calculations
```

Developer Response

Fixed in commit dfad52e.

5. Informational - Use of `balanceOf()` in vault `totalAssets()` function

It's inadvisable to rely on `balanceOf()` to get the total amount of underlying assets in a 4337-type vault, as this can open up the potential for donation attacks. There is currently no direct impact on the vault, but the change to safeguard any potential issues is simple.

Technical Details

N/A

Impact

Informational.

Recommendation

Add accounting for underlying assets in the contract rather than using `balanceOf()`.

Developer Response

Acknowledged. We have implemented `MIN_SHARES` as a minimum number of shares burned in the first deposit to the vault in order to guard against share inflation attacks including donation attacks.

Final Remarks

The audit has provided an in-depth review of Monolith smart contract codebase. The protocol introduces a novel and intricate mechanism for handling loans and redemptions, particularly through its distinction between ”paid debt” and ”free debt,” as well as the associated processes for interest accrual, collateral management, and vault interactions. Some security shortcomings were identified; however, the contracts were well-implemented overall. The protocol’s design, allowing users to opt into ”free debt” status, combined with the current redemption process, lacking scaled fees or cooldowns, presents a risk of complex and unpredictable interactions with arbitrageurs and secondary markets. While no direct attacks from this specific dynamic were found during this review, the potential for sophisticated arbitrage or economic manipulation exists. It is strongly recommended that the team conduct extensive financial modeling and stress-testing of these mechanics. This should cover scenarios such as mass opt-ins and outs of free debt, redemption spikes, and potential feedback loops to ensure stability and fairness against strategic exploitation.