Satoshi Protocol Security Review

Reviewer
Bill Hsu (@hibillh)

Draft: July 1, 2024
Finalized: July 9, 2024

1 Executive Summary

Over the course of 21 days in total, Satoshi Protocol engaged with Bill Hsu to review Satoshi Protocol.

A total of 13 issues have been found with Satoshi Protocol.

Repository	Commit
Satoshi Protocol	e74018a4a587d98205575c3c6da9b0ff3bd01551

Summary

Type of Project	CDP Protocol
Timeline	June 10, 2024 - June 30, 2024
Methods	Manual Review

Total Issues

High Risk	4
Medium Risk	3
Low Risk	3
Informational	3

Contents

Disclaimer: This security review is not a guarantee against hacking. This is the result of a review within a time range based on specific commits.

2 Findings

2.1 High Risk

2.1.1 Forcing the system into recovery mode to liquidate positions with previously healthy collateral ratios

Severity: High

Context: `LiquidationManager.sol#L397-L423`

Description:

The main issue lies in the socialize and recovery mechanism. Simply transferring the debt and collateral of a position to other users does not change the TCR. However, before redistribution, 0.5% of the collateral is deducted as a liquidator reward, causing the TCR to decrease after redistribution.

```
    /**
     * @dev Liquidate a trove without using the stability pool. All debt and
     ↪  collateral
     *          are distributed porportionally between the remaining active
     ↪  troves.
     */
    function _liquidateWithoutSP(ITroveManager troveManager, address _borrower)
        internal
        returns (LiquidationValues memory singleLiquidation)
    {
        uint256 pendingDebtReward;
        uint256 pendingCollReward;

        (singleLiquidation.entireTroveDebt, singleLiquidation.entireTroveColl,
         ↪  pendingDebtReward, pendingCollReward) =
            troveManager.getEntireDebtAndColl(_borrower);

        singleLiquidation.collGasCompensation =
         ↪  _getCollGasCompensation(singleLiquidation.entireTroveColl);
        singleLiquidation.debtGasCompensation = DEBT_GAS_COMPENSATION;
        troveManager.movePendingTroveRewardsToActiveBalances(pendingDebtReward,
         ↪  pendingCollReward);

        singleLiquidation.debtToOffset = 0;
        singleLiquidation.collToSendToSP = 0;
        singleLiquidation.debtToRedistribute =
         ↪  singleLiquidation.entireTroveDebt;
        singleLiquidation.collToRedistribute =
         ↪  singleLiquidation.entireTroveColl -
         ↪  singleLiquidation.collGasCompensation;

        troveManager.closeTroveByLiquidation(_borrower);

        return singleLiquidation;
    }
```

There are two conditions that trigger redistribution according to the protocol's design:

1. When a position's collateral ratio is below 100%.
2. When the SAT in the Stability Pool is insufficient for liquidation.

An attack can be constructed to liquidate a user with a Collateral Ratio below 150% by triggering a redistribution:

1. The attacker opens a large position with an MCR collateral ratio through a

flash loan, lowering the protocol's TCR to 150%.

2. The attacker liquidates a position with a collateral ratio below 100%, causing the TCR to fall below 150%.
3. The attacker liquidates other users with collateral ratios below 150%. After liquidation, the TCR might return to above 150%.

By combining the two conditions that trigger redistribution, an attack can be constructed to liquidate **ALL USERS** with a Collateral Ratio below 150% in a single transaction:

1. The attacker opens multiple positions with an MCR collateral ratio.
2. The attacker liquidates a position with a collateral ratio below 100%, causing the positions opened in step 1 to fall below MCR.
3. The attacker liquidates their own positions from step 1 until the SAT in the Stability Pool is depleted.
4. The attacker opens a large position with an MCR collateral ratio through a flash loan, lowering the protocol's TCR to 150%.
5. The attacker liquidates their own positions from step 1. Since the Stability Pool is empty, this triggers redistribution, causing the TCR to fall below 150%.
6. The attacker deposits SAT into the stability pool to earn the liquidation rewards.
7. The attacker liquidates other users with collateral ratios below 150%. After liquidation, the stability pool would be empty again, and the TCR might return to above 150%.
8. Repeat from step 4.

It's important to note that this method incurs a loss equal to 10% of the SAT value originally in the Stability Pool but earns liquidation rewards ($Debt \times (MCR - 100\%)$).

A position with CR < 100% is rare due to the need for a 10% price fluctuation in a short time. Alternatively, another method can be used:

1. In Transaction 1:
    1. The attacker opens multiple positions with an MCR collateral ratio.

2. Wait for next block (Or sandwitch an oracle price update, or liquidate a trove with bad debt).
3. In Transaction 2:
    1. After 12 seconds, interest accrues on the debt, causing the positions opened in Transaction 1 to fall below MCR.
    2. The attacker liquidates their own positions from Transaction 1 until the SAT in the Stability Pool is depleted.
    3. The attacker opens a large position with an MCR collateral ratio through a flash loan, lowering the protocol's TCR to 150%.
    4. The attacker liquidates their own positions from Transaction 1. Since the Stability Pool is empty, this triggers redistribution, causing the TCR to fall below 150%.
    5. The attacker deposits SAT into the stability pool to earn the liquidation rewards.
    6. The attacker liquidates other users with collateral ratios below 150%. After liquidation, the stability pool would be empty again, and the TCR might return to above 150%.
    7. Repeat from step 3.

Although this attack is feasible, the state of the protocol can make the attack unprofitable. The reasons include:

- Borrowing fees - Even the base rate has decayed, there is still a borrowing fee of 0.5%.
- Loss from emptying the Stability Pool - A 10% loss.
- Lack of positions with Collateral Ratio below 150% in the protocol - This means there's a shortage of positions that could be liquidated for rewards.

These factors combined make the attack economically unviable under certain conditions, despite its technical feasibility.

However, let's consider the following scenario:

- Protocol TCR is 200%
- Total debt in the protocol is 20M SAT
- Total collateral value is 40M USD

- Debt of positions with Collateral Ratio below 150% is 10M SAT
- Stability Pool contains 5M SAT

In this case, we can estimate the attack costs as:

- Borrowing fee = 25,000,000 * 0.5% = 125,000 USD
- Cost of emptying the Stability Pool = 5M * 0.1 = 500,000 USD

The potential profits would be:

- Liquidation rewards from the Stability Pool: 10M * 0.1 = 1M USD
- Collateral gas compensation: 10M * 0.5% = 50,000 USD

Under these circumstances, an attacker would have an economic incentive to launch the attack. It's important to note that if this attack were coordinated by Stability Pool depositors, the cost of emptying the Stability Pool could be eliminated from the attack expenses.

This example demonstrates that any user opening a position with a CR below 150% in the future becomes a potential target. The profitability of the attack depends on the protocol's state variables, which can change over time, potentially creating favorable conditions for an attacker.

Recommendation:

1. When the socialize mechanism is triggered for redistribution, set the collateral gas compensation to 0 to avoid a decrease in TCR after liquidation.
2. Increase the CR requirement to 150% when opening a position.
3. Implement a grace period after entering Recovery Mode before allowing position liquidations.

Project: Acknowledged.

2.1.2 Bypassing redemption fee by increasing debt through flash loan in recovery mode

Severity: High

Context: `TroveManager.sol#L435-L461`

Description:

When users perform redemptions, they are charged a redemption fee. This fee is calculated by multiplying the redemption amount by the redemption rate. The redemption rate consists of a decayed base rate plus a variable component based on the proportion of the redemption size to the total debt.

```
    /*
     * This function has two impacts on the baseRate state variable:
     * 1) decays the baseRate based on time passed since last redemption or
     ↪  debt borrowing operation.
     * then,
     * 2) increases the baseRate based on the amount redeemed, as a proportion
     ↪  of total supply
     */
    function _updateBaseRateFromRedemption(uint256 _collateralDrawn, uint256
    ↪  _price, uint256 _totalDebtSupply)
        internal
        returns (uint256)
    {
        uint256 decayedBaseRate = _calcDecayedBaseRate();

        /* Convert the drawn collateral back to debt at face value rate (1
        ↪  debt:1 USD), in order to get
         * the fraction of total supply that was redeemed at face value. */
        uint256 redeemedDebtFraction = (_collateralDrawn * _price) /
        ↪  _totalDebtSupply;

        uint256 newBaseRate = decayedBaseRate + (redeemedDebtFraction / BETA);
        newBaseRate = SatoshiMath._min(newBaseRate, DECIMAL_PRECISION); // cap
        ↪  baseRate at a maximum of 100%

        // Update the baseRate state variable
        baseRate = newBaseRate;
        emit BaseRateUpdated(newBaseRate);

        _updateLastFeeOpTime();

        return newBaseRate;
    }
```

In other words, the smaller the redemption amount is relative to the overall debt, the cheaper the redemption fee becomes. Users can exploit this feature by borrowing a large amount of SAT before redemption, increasing the system's total debt. They can then perform the redemption, and afterwards close their trove to repay the borrowed debt, thus reducing the redemption fee they need to pay.

Under normal circumstances, the fee reduction achievable through this method is limited because borrowing a large amount of SAT would incur substantial borrowing fees. However, according to the protocol's design, when the system enters Recovery Mode, no borrowing fees are charged on loans, thereby enabling the exploitation of this vulnerability.

Any user wanting to perform a redemption can reduce their redemption fee through the following steps:

1. Use a flash loan to borrow a large amount of collateral, then use it to borrow a large amount of SAT
2. Perform the redemption operation
3. Close the position opened in Step 1, then repay the flash loan

Recommendation:

1. Charge borrowing fees during recovery mode

Charge borrowing fees even when the system is in Recovery Mode. This would disincentivize large-scale borrowing for the purpose of manipulating the redemption rate.

2. Use parameters independent of total debt to calculate redemption rate in recovery mode

During Recovery Mode, modify the redemption rate calculation to use parameters that are not related to the total debt. This would make the redemption rate resistant to manipulation through large-scale borrowing.

3. Use time weighted total debt to calculate redemption rate in recovery mode

Project: Fixed in PR #3.

2.1.3 G would be overwritten by deposit and withdraw

Severity: High

Context: `StabilityPool.sol#L218-L238`

Description:

In the Stability Pool, `G` is used to track the amount of OSHI rewards distributed. However, the `_updateSnapshots` function, which is executed in both `provideToSP`

and `withdrawFromSP`, updates the user's snapshot `G` to the latest `G` value. This leads to a reduction in the amount of rewards users can claim.

```
    function provideToSP(uint256 _amount) external {
        require(!SATOSHI_CORE.paused(), "Deposits are paused");
        require(_amount > 0, "StabilityPool: Amount must be non-zero");

        _triggerOSHIIssuance();

        _accrueDepositorCollateralGain(msg.sender);

        uint256 compoundedDebtDeposit = getCompoundedDebtDeposit(msg.sender);

        debtToken.sendToSP(msg.sender, _amount);
        uint256 newTotalDebtTokenDeposits = totalDebtTokenDeposits + _amount;
        totalDebtTokenDeposits = newTotalDebtTokenDeposits;
        emit StabilityPoolDebtBalanceUpdated(newTotalDebtTokenDeposits);

        uint256 newDeposit = compoundedDebtDeposit + _amount;
        accountDeposits[msg.sender] = AccountDeposit({amount:
        ↪  uint128(newDeposit), timestamp: uint128(block.timestamp)});

        _updateSnapshots(msg.sender, newDeposit);
        emit UserDepositChanged(msg.sender, newDeposit);
    }
```

Recommendation:

Call `_accrueRewards` before updating a user's snapshot values.

Project: Fixed in PR #3.

2.1.4 OSHI issuance is not triggered during debt offset

Severity: High

Context: `StabilityPool.sol#L285-L302`

Description:

The `offset` function in the `StabilityPool` is executed when liquidations are performed using funds from the Stability Pool. However, the OSHI issuance is not triggered during this process. If the liquidation doesn't deplete the pool entirely, this doesn't cause an issue. However, if the pool is emptied, certain pool parameters are reset, and the epoch count is incremented. This results in Stability Pool depositors being unable to receive their OSHI rewards for that period.

```
function _offset(IERC20 collateral, uint256 _debtToOffset, uint256
↪  _collToAdd) internal {
    require(msg.sender == address(liquidationManager), "StabilityPool:
    ↪  Caller is not Liquidation Manager");
    uint256 idx = indexByCollateral[collateral];
    idx -= 1;

    uint256 totalDebt = totalDebtTokenDeposits; // cached to save an SLOAD
    if (totalDebt == 0 || _debtToOffset == 0) {
        return;
    }

    (uint256 collateralGainPerUnitStaked, uint256 debtLossPerUnitStaked) =
        _computeRewardsPerUnitStaked(_collToAdd, _debtToOffset, totalDebt,
        ↪  idx);

    _updateRewardSumAndProduct(collateralGainPerUnitStaked,
    ↪  debtLossPerUnitStaked, idx); // updates S and P

    // Cancel the liquidated Debt debt with the Debt in the stability pool
    _decreaseDebt(_debtToOffset);
}
```

Recommendation:

Call `_triggerOSHIIssuance` before `_updateRewardSumAndProduct` in `_offset`.

Project: Fixed in PR #3.

2.2 Medium Risk

2.2.1 Inaccurate base rate decay due to time rounding

Severity: Medium

Context: `TroveManager.sol#L516-L524`

Description:

1. The `_calcDecayedBaseRate` function uses minutes as its time unit for decay calculations.
2. If fee operations occur at intervals just under a full minute (e.g., every 1 minute and 59 seconds), each operation will be rounded down to 1 minute.

3. This rounding error can accumulate, potentially doubling the effective half-life of the base rate decay.
4. As a result, the base rate may decay much slower than intended, affecting borrowing and redemption fees.

The root cause is in the `_updateLastFeeOpTime` function, which updates `lastFeeOperationTime` to the current timestamp without accounting for partial minutes.

Recommendation:

update the `_updateLastFeeOpTime` function to account for partial minutes:

```
lastFeeOperationTime = lastFeeOperationTime + (block.timestamp -
↪  lastFeeOperationTime) / SECONDS_IN_ONE_MINUTE * SECONDS_IN_ONE_MINUTE;
```

This change ensures that `lastFeeOperationTime` is updated more accurately, preventing the accumulation of rounding errors and maintaining the intended decay rate of the base fee.

Project: Fixed in PR #3.

2.2.2 Inaccurate reward calculation in StabilityPool's `claimableReward` Function

Severity: Low

Context: `StabilityPool.sol#L469-L495`

Description:

The logic error in the `claimableReward` function of the `StabilityPool` contract manifests in two areas.

Firstly, when the total debt in the pool is zero, typically following a liquidation that empties the pool, the function only returns the `storedPendingReward`. This approach overlooks any unclaimed rewards that haven't been stored yet, potentially underestimating the user's total claimable amount.

Secondly, the calculation of `marginalOSHIGain` fails to consider whether the current epoch matches the epoch recorded in the user's snapshot. This oversight can lead to incorrect calculations when epoch changes have occurred, as the `marginalOSHIGain` should be zero if the epochs don't match.

```
    function claimableReward(address _depositor) external view returns
    ↪  (uint256) {
        uint256 totalDebt = totalDebtTokenDeposits;
        uint256 initialDeposit = accountDeposits[_depositor].amount;

        if (totalDebt == 0 || initialDeposit == 0) {
            return storedPendingReward[_depositor];
        }
        uint256 oshiNumerator = (_OSHIIssuance() * DECIMAL_PRECISION) +
        ↪  lastOSHIError;
        uint256 oshiPerUnitStaked = oshiNumerator / totalDebt;
        uint256 marginalOSHIGain = oshiPerUnitStaked * P;

        Snapshots memory snapshots = depositSnapshots[_depositor];
        uint128 epochSnapshot = snapshots.epoch;
        uint128 scaleSnapshot = snapshots.scale;
        uint256 firstPortion;
        uint256 secondPortion;
        if (scaleSnapshot == currentScale) {
            firstPortion = epochToScaleToG[epochSnapshot][scaleSnapshot] -
            ↪  snapshots.G + marginalOSHIGain;
            secondPortion = epochToScaleToG[epochSnapshot][scaleSnapshot + 1] /
            ↪  SCALE_FACTOR;
        } else {
            firstPortion = epochToScaleToG[epochSnapshot][scaleSnapshot] -
            ↪  snapshots.G;
            secondPortion = (epochToScaleToG[epochSnapshot][scaleSnapshot + 1]
            ↪  + marginalOSHIGain) / SCALE_FACTOR;
        }

        return storedPendingReward[_depositor]
            + (initialDeposit * (firstPortion + secondPortion)) / snapshots.P /
            ↪  DECIMAL_PRECISION;
    }
```

Recommendation:

1. Include unclaimed rewards when `totalDebt` is zero:

```
if (totalDebt == 0 || initialDeposit == 0) {
  return storedPendingReward[_depositor] + _claimableReward(_depositor);
}
```

2. Add an epoch check before calculating `marginalOSHIGain`:

```
uint256 marginalOSHIGain = (epochSnapshot == currentEpoch) ? oshiPerUnitStaked
↪  * P : 0;
```

Project: Fixed in PR #3.

2.2.3 Inaccurate OSHI reward calculation due to interest accrual discrepancy

Severity: Low

Context: `TroveManager.sol#L940-L974`

Description:

The issue stems from two main factors:

1. The `_updateRewardIntegral` function uses the current total debt (including accrued interest) to calculate rewards.
2. The `_applyPendingRewards` function uses the user's debt before interest accrual to calculate their individual rewards.

```
    // Add the borrowers's coll and debt rewards earned from redistributions,
    ↪  to their Trove
    function _applyPendingRewards(address _borrower) internal returns (uint256
    ↪  coll, uint256 debt) {
        Trove storage t = troves[_borrower];
        if (t.status == Status.active) {
            uint256 troveInterestIndex = t.activeInterestIndex;
            uint256 supply = totalActiveDebt;
            uint256 currentInterestIndex = _accrueActiveInterests();
            debt = t.debt;
            uint256 prevDebt = debt;
            coll = t.coll;
            // We accrued interests for this trove if not already updated
            if (troveInterestIndex < currentInterestIndex) {
                debt = (debt * currentInterestIndex) / troveInterestIndex;
                t.activeInterestIndex = currentInterestIndex;
            }

            if (rewardSnapshots[_borrower].collateral < L_collateral) {
                // Compute pending rewards
                (uint256 pendingCollateralReward, uint256 pendingDebtReward) =
                ↪  getPendingCollAndDebtRewards(_borrower);

                // Apply pending rewards to trove's state
                coll = coll + pendingCollateralReward;
                t.coll = coll;
                debt = debt + pendingDebtReward;

                _updateTroveRewardSnapshots(_borrower);

                _movePendingTroveRewardsToActiveBalance(pendingDebtReward,
                ↪  pendingCollateralReward);
            }
            if (prevDebt != debt) {
                t.debt = debt;
            }
            _updateIntegrals(_borrower, prevDebt, supply);
        }
        return (coll, debt);
    }
```

This discrepancy can lead to users receiving fewer OSHI rewards than they should, especially if they haven't interacted with the contract for a long time, allowing interest to accrue without updating their position.

Recommendation:

1. Modify `_applyPendingRewards` to use the debt amount after interest accrual when calculating rewards.
2. Ensure that `_updateRewardIntegral` uses the total debt including the most recent interest accrual.
3. Adjust the reward integral when collecting interest:

```
uint256 adjustmentFactor = INTEREST_PRECISION / (INTEREST_PRECISION
↪  + interestFactor);
rewardIntegral = rewardIntegral * adjustmentFactor;
```

4. For a user, the amount of rewards in a period can be calculated this way:

```
(currentDebt * currentIntegral - prevDebt * prevIntegral) / 1e18
```

5. Update `claimableReward` accordingly.

Project: Acknowledged.

2.3 Low Risk

2.3.1 Inaccurate total collateral ratio (TCR) calculation due to unaccounted collateral gas compensation

Severity: Low

Context: LiquidationManager.sol#L204-L308

Description:

The vulnerability lies in the TCR calculation process within the `batchLiquidateTroves` and `liquidateTroves` functions of the `LiquidationManager` contract. These functions use `entireSystemColl` and `entireSystemDebt` to compute the system's TCR, which is a critical metric for assessing the protocol's health and determining when liquidations should occur.

After each trove liquidation, these values are updated to reflect changes in the system's collateral and debt. However, an oversight in this update process fails to account for the `collGasCompensation` paid to the liquidator. Consequently, the `entireSystemColl` value used in TCR calculations remains artificially inflated. This discrepancy leads to a calculated TCR that is higher than the actual TCR of the system.

```
        if (troveIter < length && troveCount > 1) {
            // second iteration round, if we receive a trove with ICR > MCR and
            ↪  need to track TCR
            (uint256 entireSystemColl, uint256 entireSystemDebt) =
            ↪  borrowerOperations.getGlobalSystemBalances();
            entireSystemColl -= totals.totalCollToSendToSP *
            ↪  troveManagerValues.price;
            entireSystemDebt -= totals.totalDebtToOffset;
```

Recommendation:

Update the TCR calculation to account for the `collGasCompensation`.

Project: Fixed in PR #3.

2.3.2 Potential for incorrect LP token price calculation due to oracle decimal mismatch

Severity: Low

Context: `PriceFeedUniswapV2LP.sol#L41-L101`

Description:

The current implementation in both `fetchPrice` and `fetchPriceUnsafe` functions calculates the LP token price using the formula:

```
2 * sqrt(r0 * r1 * p0 * p1) / totalSupply
```

This calculation is correct when the following condition is met:

```
oracle1.decimals() + oracle2.decimals() == _decimals * 2
```

However, if either of the oracles has a different number of decimals than the contract's `_decimals`, the calculation will likely produce an incorrect result.

Recommendation:

To make the calculation correct for all cases, regardless of the oracle decimals:

1. Scale oracle prices to a common base (e.g., 18 decimals) before calculation:

```
uint256 price1 = oracle1.fetchPrice() * 10**(18 - oracle1.decimals());
uint256 price2 = oracle2.fetchPrice() * 10**(18 - oracle2.decimals());
```

2. Perform the calculation using these normalized values.

3. Scale the result to the contract's desired decimal places:

```
uint256 lpPrice = lpPrice18 / 10**(18 - _decimals);
```

Project: Fixed in PR #3.

2.3.3 Precision loss in LP token price calculation due to intermediate division

Severity: Low

Context: PriceFeedUniswapV2LP.sol#L516-L524

Description:

The current implementation calculates the LP token price using:

```
FixedPointMathLib.sqrt(
    r0.mulWadDown(r1)
    .mulWadDown(oracle1.fetchPrice())
    .mulWadDown(oracle2.fetchPrice())
).mulDivDown(2e27, IUniswapV2Pair(pair).totalSupply())
```

This approach has several issues:

1. The use of `mulWadDown` performs division by 1e18 after each multiplication, which can lead to significant precision loss, especially when dealing with small numbers.
2. In extreme cases, such as when `r0 * r1 < 1e18`, the intermediate result becomes zero due to integer division, causing the entire calculation to return zero.

Recommendation:

Implement the calculation using two separate square roots to avoid intermediate division and maintain precision:

```
uint256 lpPrice =
    2 * FixedPointMathLib.sqrt(r0 * price0) * FixedPointMathLib.sqrt(r1 *
    ↪  price1)
    / IUniswapV2Pair(pair).totalSupply();
```

Project: Fixed in PR #3.

2.4 Informational

2.4.1 Inconsistency between DebtToken contract implementation and documentation

Severity: Informational

Context: DebtToken.sol#L21-L21

Description:

In the DebtToken contract, there is a discrepancy between the contract's implementation and its documentation. The contract comment states that it is "Non-upgradeable", but the contract itself inherits from UUPSUpgradeable and implements upgrade-related functions, indicating that it is designed to be upgradeable.

Recommendation:

1. If the contract is intended to be upgradeable, update the comment to reflect this.
2. If the contract is not intended to be upgradeable, remove the upgrade-related implementations and inheritance.

Project: Fixed in PR #3.

2.4.2 Redundant storage of satoshi core address

Severity: Informational

Context: Factory.sol#L31-L31

Description:

In the Factory contract, there is a redundant storage of the Satoshi Core address. The contract inherits from SatoshiOwnable, which already declares a storage variable for the Satoshi Core address, yet the Factory contract declares its own storage variable for the same purpose.

```
contract Factory is IFactory, SatoshiOwnable, UUPSUpgradeable {
    ISatoshiCore public satoshiCore;
```

Recommendation:

Remove the satoshiCore declaration from the Factory contract.

Project: Fixed in PR #3.

2.4.3 Inconsistency between comment and implementation of `startTime` variable

Severity: Informational

Context: SatoshiCore.sol#L32-L38

Description:

The comment for the startTime variable states:

```
// System-wide start time, rounded down the nearest epoch week.
// Other contracts that require access to this should inherit `SystemStart`.
uint256 public immutable startTime;
```

This comment suggests that `startTime` should be rounded down to the nearest epoch week. However, in the constructor, `startTime` is simply set to the current block timestamp without any rounding:

```
constructor(address _owner, address _guardian, address _feeReceiver, address
↪  _rewardManager) {
  owner = _owner;
  startTime = block.timestamp;
  // ... other initializations
}
```

This implementation does not align with the comment's description, as it doesn't perform any rounding to the nearest epoch week.

Recommendation:

1. Implement the rounding as described in the comment:

```
uint256 public constant SECONDS_IN_WEEK = 7 * 24 * 60 * 60;
constructor(...) {
    // ...
    startTime = (block.timestamp / SECONDS_IN_WEEK) * SECONDS_IN_WEEK;
    // ...
}
```

2. Or, if rounding is not actually required, update the comment to accurately reflect the implementation.

Project: Fixed in PR #3.