CD SECURITY

AUDIT REPORT

Satoshi Finance

March 2024

Introduction

A time-boxed security review of the **Satoshi Finance** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Satoshi Finance**

Satoshi Finance operates as a decentralized stablecoin protocol, forked from Liquify. It enables users to mint `btUSD`, a USD-pegged token, utilizing `BTCB` as collateral on the BNB Smart Chain.

Within the ecosystem, three key contracts - `BorrowerOperations.sol`, `TroveManager.sol`, and `StabilityPool.sol` - serve as the interface for users, hold the user-facing public functions and contain most of the internal system logic.

Notable adaptations from the original Liquify codebase include:

- Elimination of `SortedList.sol` (removal of Trove ordering)
- Alteration of redemption behavior, redistributing redeemed debt and collateral to all Troves
- Introduction of a new ecosystem participant, the `scavenger` tasked with Trove closure assistance
- Implementation of a premium membership for growth token staking
- Integration of ERC3156 feature-compliant support for collateral flash loans
- Incorporation of Binance Oracle functionality in `PriceFeed.sol` as a secondary failover

Full Documentation of Satoshi Finance

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - d50d9bedc876aa3b1d11b1c57ec8f31b90a3a533**

Scope

The following smart contracts were in scope of the audit:

- `./Dependencies/LiquityBase.sol`
- `./SATO/CommunityIssuance.sol`
- `./SATO/SATOStaking.sol`
- `./ActivePool.sol`
- `./BorrowerOperations.sol`
- `./CollSurplusPool.sol`
- `./DefaultPool.sol`
- `./PriceFeed.sol`
- `./StabilityPool.sol`
- `./TroveManager.sol`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 1 issues
- Low: 1 issues

Findings Summary

ID	Title	Severity
[M-01]	Decay interval can be extended	Medium
[L-01]	User will have to unstake before go premium	Low

Detailed Findings

[M-01] Decay interval can be extended

Severity

Impact: Medium

Likelihood: Medium

Description

`decayBaseRateFromBorrowing()` calls `_calcDecayedBaseRate()` to calculate the decayed base rate based how many minutes elapsed since last recorded `lastFeeOperationTime`:

```
function decayBaseRateFromBorrowing() external override {
        _requireCallerIsBorrowerOperations();

        uint decayedBaseRate = _calcDecayedBaseRate();
        assert(decayedBaseRate <= DECIMAL_PRECISION);  // The baseRate can
decay to 0

        baseRate = decayedBaseRate;
        emit BaseRateUpdated(decayedBaseRate);

        _updateLastFeeOpTime();
    }

   function _calcDecayedBaseRate() internal view returns (uint) {
        uint minutesPassed = _minutesPassedSinceLastFeeOp();
        uint decayFactor = LiquityMath._decPow(MINUTE_DECAY_FACTOR,
minutesPassed);

        return baseRate.mul(decayFactor).div(DECIMAL_PRECISION);
    }

 function _minutesPassedSinceLastFeeOp() internal view returns (uint) {
        return
(block.timestamp.sub(lastFeeOperationTime)).div(SECONDS_IN_ONE_MINUTE);
 }
```

`decayBaseRateFromBorrowing()` then calls `_updateLastFeeOpTime()` to set `lastFeeOperationTime` to the current time if at least 1 minute has passed:

```
 function _updateLastFeeOpTime() internal {
        uint timePassed = block.timestamp.sub(lastFeeOperationTime);

        if (timePassed >= SECONDS_IN_ONE_MINUTE) {
            lastFeeOperationTime = block.timestamp;
            emit LastFeeOpTimeUpdated(block.timestamp);
        }
    }
```

The problem with such an update of `lastFeeOperationTime` is, if 1.999 minutes has passed, the base rate will only decay for 1 minute, at the same time, 1.999 minutes will be added to `lastFeeOperationTime`. In other words, in the worst scenario, for every 1.999 minutes, the base rate will only decay for 1 minute. Therefore, the base rate will decay more slowly then expected.

The borrowing base rate is very fundamental to the whole protocol. Any small deviation is accumulative. In the worse case, the decay speed will slow down by half; on average, it will be 0.75 slower.

Recommendations

Using the effective elapsed time that is consumed by the model so far to revise `lastFeeOperationTime`.

```
 function _updateLastFeeOpTime() internal {
        uint timePassed = block.timestamp.sub(lastFeeOperationTime);

        if (timePassed >= SECONDS_IN_ONE_MINUTE) {
-            lastFeeOperationTime = block.timestamp;
+            lastFeeOperationTime +=
_minutesPassedSinceLastFeeOp()*SECONDS_IN_ONE_MINUTE;
            emit LastFeeOpTimeUpdated(block.timestamp);
        }
    }
```

[L-01] User will have to unstake before go premium

The users can decide to stake premium, lock their tokens forever, and enjoy some extra benefits. If an user stake tokens over time, it is not possible to directly go premium if there are enough accumulated staked tokens in his balance because the `goPremiumStaking` checks only if there are enough tokens inside the `balanceOf(msg.sender)`. This results in unnecessary additional call and waste of gas. Consider adding a check if the already staked tokens are `>= PREMIUM_STAKING`.