Satoshi Finance

Smart Contract Security Assessment

Feb. 26, 2024

ABSTRACT

Dedaub was commissioned to perform a security audit of the Satoshi Finance protocol, which is a fork of Liquity to launch on Binance. The audit was over smart contract code. Both the code and the accompanying artifacts (i.e., test-suite, documentation) adhere to high professional standards.

BACKGROUND

Satoshi Finance attempts to change how Liquity's redemption mechanism works by making redemptions "socialized". During a debt redemption, the protocol does not prioritize the troves with the lowest individual collateralization ratio (ICR) but instead distributes the redeemed debt across all active troves. This change has the additional effect of removing the requirement of maintaining a sorted ordering of all active troves based on their ICR, potentially simplifying the implementation of the protocol.

Additionally, Satoshi Finance introduces the following modifications to the original Liquity codebase:

- The Active Pool now offers an ERC3156-compliant flashloan implementation.
- Protocol stakers can opt for a premium membership that will provide them with improved fee rates within the protocol.
- A new ecosystem party, scavengers, will be responsible for closing down troves whose debt gets lower than the required minimum because of how redemptions are now "socialized".
- When a trove gets created, no debt is reserved as a "gas compensation" for a potential liquidation of the trove

SETTING & CAVEATS

This audit report mainly covers the contracts of the public repository https://github.com/Satoshi-Finance/sato-dev of the protocol at commit `787bac5aa5754b78412457791e6280b07150d191`. We performed the audit as a "delta" audit, inspecting the differences relative to commit `ce7f382e2be3c360c38d6ad838b6ab658b588e4e` of the Liquity codebase.

2 auditors worked on the codebase for 5 days on the following contracts:

```
contracts/
├── ActivePool.sol
├── BTUSDToken.sol
├── BorrowerOperations.sol
├── CollSurplusPool.sol
├── DefaultPool.sol
├── Dependencies/
│   └── LiquityBase.sol
├── LPRewards/
│   └── Unipool.sol
├── PriceFeed.sol
├── SATO/
│   ├── CommunityIssuance.sol
│   ├── LockupContract.sol
│   ├── LockupContractFactory.sol
│   ├── SATOStaking.sol
│   └── SATOToken.sol
├── StabilityPool.sol
└── TroveManager.sol
```

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

The above is particularly a concern in a delta audit, such as the present one. Auditing only the delta of the functionality compared to the original baseline (the Liquity code) has some threats, especially threats of omission: it is relatively easier to see via auditing that added code is erroneous, but much harder to discover that a certain key update is missing, e.g., for correct accounting or maintaining a consistent protocol state. This burden falls on thorough testing: the validity of the protocol's accounting can be fully ascertained only by the existence of appropriate test cases. The Satoshi codebase extends the (already extensive) Liquity tests, yet still we remind of the need for vigilance in very high test coverage of possible code conditions.

PROTOCOL-LEVEL CONSIDERATIONS

Some design decisions have economic consequences that differ from the tradeoffs of the underlying Liquity system. We bring them up for awareness.

ID	Description	STATUS
P1	Updated condition for exiting Stability Pool	**INFO**

The system features a different condition under which withdrawals from the Stability Pool were halted. Liquity used to not allow withdrawals from the Stability Pool if there were any liquidatable troves. This is not possible in the Satoshi system, since the troves are no longer sorted by their collateralization ratio (ICR), so the ones that are liquidatable cannot be found on-chain. The updated condition is:

StabilityPool:1027

```
function _requireSystemTCR() internal {
    uint price = priceFeed.fetchPrice();
    uint tcr = troveManager.getTCR(price);
    require(tcr > CCR, "StabilityPool: Cannot withdraw while TCR is low");
}
```

Thus, nobody can exit the SP even when there is nothing to be gained nor lost, if TCR <= 130%. One consequence, for instance, may be to deter investors from entering the SP. Since `tcr < CCR` (i.e., the system being in "recovery mode") should be rare and temporary, the concern is moderated.

P2	Premium staking results in permanently locking tokens, cannot be reversed	**INFO**

A premium SATO staker locks-in the premium fee forever, with no possibility of withdrawing it. Accordingly, once a staker is a premium staker, they remain forever.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants

	can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Stability Pool staker can front-run liquidations	**LARGELY RESOLVED (commits d50d9be, 753e2a8)**

The different condition for withdrawing from the Stability Pool (item P1), aided by the ability to easily circumvent the timelock (item M1) result in Stability Pool stakers being able to front-run large liquidations. Specifically, the motivation for the Liquity mechanism for preventing SP withdrawals was (from the README): "*To prevent potential loss evasion by depositors, withdrawals from the Stability Pool are suspended when there are liquidable Troves with ICR < 110% in the system.*"

In Satoshi finance, this threat is not prevented. If the TCR is > 130% (CCR) and a large liquidation of a trove with extremely low ICR (say, 80%) comes in, then the SP staker can front-run it and avoid the loss.

The issue can be significantly mitigated, although not completely eliminated, by addressing the timelock problem (item M1).

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Timelock can be subverted	**RESOLVED (commits d50d9be, 753e2a8)**

The timelock mechanism is not effective. Function `requestWithdrawFromSP` can be called well in advance of the actual withdraw (arbitrarily early). In this way, a Stability Pool depositor can do a `requestWithdrawFromSP` for their full deposited amount as soon as they join. Within 30 mins, withdrawals are enabled and the permission to do a full withdrawal stands forever. When the withdrawal actually happens, the amount supplied to function `withdrawFromSP` is ignored.
`StabilityPool:398`

```
function withdrawFromSP(uint _amount) external override {
  if (_amount != 0) {
    _requireSystemTCR();

    // check withdrawal delay
    uint _existReqAmount = withdrawReqAmount[msg.sender];
    require(_existReqAmount > 0,
            "StabilityPool: no valid existing withdrawal request");

    …
    _amount = _existReqAmount;
  }
  …
  (uint compoundedDebtDeposit, uint debtToWithdraw) =
     _capWithdrawAmount(msg.sender, _amount);
```

Instead, the current full deposit amount (adjusted for rewards and redemption amounts) is compared with the earlier withdraw request and the minimum is taken (in function `_capWithdrawAmount`).

`StabilityPool:495`

```
function _capWithdrawAmount(address _depositor, uint _amount) internal
view returns(uint, uint){
      uint compoundedDebtDeposit = getCompoundedDebtDeposit(_depositor);
      if (_amount == 0){
            return (compoundedDebtDeposit, _amount);
      } else {
            return (compoundedDebtDeposit,
                    LiquityMath._min(_amount, compoundedDebtDeposit));
      }
}
```

In this way, the depositor can defeat the timelock and withdraw their full deposit at any point beyond the first 30 mins. Additionally, it is easy to nullify earlier withdrawal requests (e.g., to later enable a partial withdrawal) by calling `withdrawFromSP` with a 0 amount.

LOW SEVERITY:

ID	Description	STATUS
L1	Oracle check incomplete	**RESOLVED (commit 6254c11)**

Function `PriceFeed::_backupIsBroken`, which checks the validity of the backup oracle information, is weaker than `PriceFeed::_badChainlinkResponse`.

`PriceFeed:376`

```
function _backupIsBroken(ChainlinkResponse memory _response) internal view
returns (bool) {
      // Check for response call reverted
      if (!_response.success) {return true;}
      // Check for an invalid timeStamp that is 0, or in the future
      if (_response.timestamp == 0 ||
          _response.timestamp > block.timestamp) {return true;}
      // Check for zero price
      if (_response.answer == 0) {return true;}

      return false;
}
```

Specifically, the function is missing tests for invalid `roundIds` and for `_response.answer < 0`.

We recommend adding both of these extra tests.

CENTRALIZATION ISSUES:

The protocol is highly decentralized, following the example set by Liquity. This is an excellent feature, but, as a side effect, raises the bar for correctness, since no administrator can fix, pause, evolve, or rescure the protocol. Liquity clones have been found to have accounting errors, even after years of successful operation. (See, e.g., the recent wrap up of Yeti Finance.) The accounting mathematics is simply quite complex and it is very easy for bugs of omission to creep in. Therefore, we again point out the need for extremely thorough testing, for purposes of functional correctness (not security, except indirectly).

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Parameter unnecessary	**INFO**

ActivePool:232

```
function flashFee(address token, uint256 amount) public view override
returns (uint256) {
   require(token == address(collateral), "ActivePool: collateral Only");
   return amount.mul(FLASH_FEE_BPS).div(MAX_BPS);
}
```

It is not clear why the `token` parameter needs to be supplied, if it is always forced to be equal to storage data.

ID	Description	STATUS
A2	Unnecessary conditionals	**INFO**

The code below could be simplified, by taking advantage of Solidity’s structural conformance type system.

Instead of:

ActivePool::_sendCollateralOut:132

```
if (_account == collSurplusPoolAddress){
   ICollSurplusPool(collSurplusPoolAddress).receiveCollateral(_amount);
} else if (_account == defaultPoolAddress){
   IDefaultPool(defaultPoolAddress).receiveCollateral(_amount);
} else if (_account == stabilityPoolAddress){
   IStabilityPool(stabilityPoolAddress).receiveCollateral(_amount);
}
```

one could have merely:

```
if (_account == collSurplusPoolAddress ||
    _account == defaultPoolAddress ||
    _account == stabilityPoolAddress) {
   ICollSurplusPool(collSurplusPoolAddress).receiveCollateral(_amount);
}  // All three pools support the same function signature
```

A3	Unnecessary arguments	INFO

The _activePool and _defaultPool arguments of TroveManager::_liquidateRecoveryMode are unused:

TroveManager.sol::_liquidateRecoveryMode

```
function _liquidateRecoveryMode(
      IActivePool _activePool,
      IDefaultPool _defaultPool,
      ...
)
```

Likewise, both _activePool and _defaultPool arguments of TroveManager::_applyPendingRewards are unused:

TroveManager.sol::_applyPendingRewards

```
function _applyPendingRewards(
      IActivePool _activePool,
      IDefaultPool _defaultPool,
      ...
)
```

A4	Unnecessary memory read?	**INFO**

The code below reads from memory values also stored in stack variables. Does this avoid a stack-too-deep error, or can it be optimized?

BorrowerOperations::_adjustTrove:263

```
      vars.isCollIncrease = _isCollIncrease;
      vars.collChange = _collChange;
...
      vars.newICR =
        _getNewICRFromTroveChange(vars.coll, vars.debt, vars.collChange,
                                  vars.isCollIncrease, vars.netDebtChange,
                                  _isDebtIncrease, vars.price);
      if (!vars.isCollIncrease){
...
```

(If changes are made to such code, they should be made very carefully.)

A5	Misnamed functions, variables, or messages	**INFO**

There are many functions, variables, revert messages, etc. that maintain the Liquity naming instead of the updated Satoshi naming. A non-exhaustive list includes:

- members called ETH, e.g., in DefaultPool
- all getETH functions, accordingly
- comment "// Move the ether to ..." in BorrowerOperations:194

- functions PriceFeed::_chainlinkIsFrozen and _storeChainlinkPrice, which are used for Chainlink and backup oracles alike
- StabilityPool::_sendETHGainToDepositor
- TroveManager::getPendingETHReward and getPendingLUSDDebtReward
- SatoStaking::F_ETH and F_LUSD

A6	Documentation inconsistent	INFO

The Satoshi Finance repo README file is mostly still the Liquity README. This is used for a lot of protocol documentation, which no longer applies. For instance, the definition of CCR (the critical collateralization ratio that triggers recovery mode) is listed as 150%, not 130% as defined in the code.

A7	Function does not need to be public	INFO

The function below does not seem to be called from smart contracts and can be declared external instead of public.

CommunityIssuance:131

```
function getSATOYetToIssue() public override view returns (uint256) {
      uint latestTotalSATOIssued = _getLatestIssuedSATO();
      return latestTotalSATOIssued >= SATOSupplyCap?
              0 : SATOSupplyCap.sub(latestTotalSATOIssued);
}
```

A8	Unnecessary permissioning	INFO

Even though both TroveManager and BorrowerOperations are allowed to call SATOStaking::increaseF_LUSD , the TroveManager contract never actually calls the function.

SATOStaking.sol::increaseF_LUSD:215

```
function increaseF_LUSD(uint _debtMintFee) external override {
      _requireCallerIsTroveManagerOrBO();
      ...
```

A9	Unused variable	INFO

The `amountWithFee` variable inside `ActivePool::flashLoan` is unused:
`ActivePool::flashLoan.sol:200`

```
function flashLoan(
      IERC3156FlashBorrower receiver,
      address token,
      uint256 amount,
      bytes calldata data
      ) external override returns (bool) {
      ...
      uint256 amountWithFee = amount.add(fee);
      ...
```

A10	Misspelled identifier	INFO

`StabilityPool::_getSATOGainFromSnapshotsAndGlobleG`: Globle → Global?

A11	Compiler bugs	INFO

The code is compiled with Solidity `0.6.11`. This version has some known bugs, which we do not believe affect the correctness of the contracts.

DISCLAIMER

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.