CERTORA

Formal Verification of BentoBox

Summary

This document describes the specification and verification of **BentoBox** form SushiSwap using Certora Prover. The work was undertaken from **Jan. 25 to Feb. 9 2021**. The latest commit that was reviewed and run through the Certora Prover was **0c68f2aa38cd600b5959197416fe090baa6a2a3f**.

The scope of our verification was the BentoBox contract that stores funds, handles their transfers, and supports flash loans and strategies. In addition to the BentoBox as a whole, a special focus lay on the mathematical Rebase library for representing ratios of absolute shares and a total “elastic” amount and the SushiStrategy used by the BentoBox for investing assets.

The Certora Prover proved the implementation of the BentoBox correct with respect to the formal rules written by the SushiSwap and the Certora teams. During the verification process, the Certora Prover discovered issues in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high-level specifications.

All the rules are publically available in a public GitHub repository:
https://github.com/sushiswap/bentobox/tree/master/spec

List of Main Issues Discovered

Severity: High

Issue:	Withdraw extra assets
Rules Broken:	Total assets of user
Description:	A user can withdraw an amount that, due to rounding, reduces their balance by shares that correspond to a lower amount and transfer the amount requested. One can repeat this step until no profit is left in the system.
Mitigation:	Use explicit rounding up/down depending on the context to have the system always benefit from rounding errors.

Severity: Medium

Issue:	Withdraw of profit
Rules Broken:	No change to others
Description:	In case there is profit in the BentoBox, a user can withdraw an amount that is less than one share which would be rounded down to zero share, and repeat this operation until no profit is left in the system.
Mitigation:	Withdraw of at least one share

Severity: Medium

Issue:	Lock of assets
Rules Broken:	Inverse of deposit and withdraw
Description:	Users can not withdraw and leave the BentoBox below some minimum threshold.
Mitigation:	Reduce the minimum to an insignificant amount

Severity: Medium

Issue:	Denial-of-service on flashloan
Rules Broken:	N/A
Description:	When the system has excess tokens, a valid flashloan returning the exact required amount fails.
Mitigation:	Weaken requirement to take into account the excess collateral

Severity: Low

Issue:	Inconsistent representation of total assets
Rules Broken:	Solvency
Description:	Incorrect updating of negative profit from strategy for support of future strategies.
Mitigation:	New interface for strategies

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, but provide no warranty of any kind, express or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

1. ✔ indicates the rule is formally verified on the latest commit. We write ✔* when the rule was verified on a simplified version of the code or under some additional assumptions.
2. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.
3. The syntax {p} ($C_1 \sim C_2$) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \sim$op C_2, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of RebaseLibrary

The Rebase library enables storing “shares” (also called the “base”) and conversion to “amount” (also called “elastic”) or vice-versa from “amount” to “shares”. It is the basis for many computations done in the BentoBox.

Functions

All operations in the Rebase library refer to a structure of type **Rebase** with two members:

- **uint128** elastic representing amount
- **uint128** base representing share

The following functions are called on an instance of the Rebase struct. They all take into account the ratio of base/elastic stored in the struct, some manipulate it.

1. **toBase(uint256 xElastic, bool roundUp) : uint**
   Converts the amount xElastic to a number of shares.
2. **toElastic(uint256 xBase, bool roundUp) : uint**
   Converts a number of shares, xBase, to an amount.
3. **add(uint256 xElastic, bool roundUp) : uint**
   Adds xElastic to the elastic field keeping the ratio of base/elastic the same.
4. **sub(uint256 xBase, bool roundUp) : uint**
   Subtracts xBase to the base field keeping the ratio of base/elastic the same.
5. **addElastic(uint256 xElastic) : uint**
   Adds xElastic to the elastic field without updating base.

The roundUp flag allows the user of these functions to control whether the division operations used in these functions should round up instead of down. We omit the flag in the method calls below for simplicity. If there is no further comment, we left the flag unconstrained but constant within the rule (i.e. if two calls in the rule use the flag, it will be either set or not set for both at the same time).

Properties

v is an instance of type rebase.

x is an arbitrary integer.

y ≅ z means that y and z are “almost equal”, which means that they do not deviate beyond a certain bound (see comments for each rule for what that bound is).

1. **Identity** ✔(rule: toElasticAndToBaseAreInverse*)
   The operations toElastic and toBase must be inverses of each other.

   ```
   v.toElastic(v.toBase(x)) ≅  x
   ```

   The error is bounded by the term: v.base / v.elastic + 2. In the special case where v.base or v.elastic is/are 0, v.toElastic(v.toBase(x)) can become 0 even for non-zero inputs.

   ```
   v.toBase(v.toElastic(x)) ≅  x
   ```

   The error is bounded by the term: v.elastic / v.base + 2. In the special case where v.base or v.elastic is/are 0, v.toBase(v.toElastic(x)) can become 0 even for non-zero values of x.

2. **Monotonicity** ✔ (rules: toBaseIsMonotone and toElasticIsMonotone)
   The operations toElastic and toBase must be monotone, i.e., if we increase their input value, their output value must (non-strictly, due to rounding) increase, too.

   ```
   x < y ⇒ v.toBase(x) ≤ v.toBase(y)
   ```

   ```
   x < y ⇒ v.toElastic(x) ≤ v.toElastic(y)
   ```

Verification of BentoBox

BentoBox is a vault for tokens. Stored tokens can be flash loaned. In addition, tokens can be invested in strategies. Fees and profits (positive or negative) are split between owners of tokens according to shares. This contract allows: depositing, withdrawing, transferring, flash loaning.

Functions:

1. **toShare(address token, uint amount, bool roundUp) : uint**
   Returns the amount of shares for the given amount of total, rounding up or down.

2. **toAmount(address token, uint share, bool roundUp) : uint**
   Returns the amount of tokens for the given amount of share, rounding up or down.

3. **deposit(address token, address from, address to, uint amount, uint share)**
   Deposits an amount of a token (either given in terms of elastic amount or as a number of shares), from a sender to some BentoBox account.

4. **withdraw(address token, address from, address to, uint amount, uint share)**
   Withdraws an amount of a token (either given in terms of elastic amount or as a number of shares), from a BentoBox account to some account.

5. **transfer(address token, address from, address to, uint share)**
   Transfers an (either given in terms of elastic amount or as a number of shares) between two BentoBox accounts.

6. **flashLoan(address borrower, address receiver, address token, uint amount)**
   Make a flash-loan.

7. **harvest(address token, bool balance, uint maxChangeAmount)**
   Account for and possibly withdraw profit (possibly negative) from the strategy for a given token.

Properties:

1. **State integrity ✔** (rule: integrityOfTotalShare)
   The sum of the number of shares on all user accounts must match the number of total shares.

   ```
   v := totals[token]
   ```

   $v.base = \sum_{address\ a}$ `balanceOf[token][a]`

2. **Solvency ✔** (rule: solvency)
   The internal representation of total assets is smaller or equals to actual assets, due to transfers to the BentoBox that are not accounted for.

   ```
   token.balanceOf(bentoBox) + strategyData[token].balance ≥
   totals[token].elastic
   ```

3. **Total Assets of user ✔** (rule: preserveTotalAssetsOfUser)
   For every token and every operation, the number of total assets within the BentoBox and outside the BentoBox is preserved,

   AND

   If other users transfer or deposit to user *a*, then the balance of *a* should only go up.

   ```
   { v = token.balanceOf(a) + toAmount(token, balanceOf[token][a]) }

         op

   { v <= token.balanceOf(a) +

           toAmount(token, balanceOf[token][a]) }
   ```

   The flashloan operation is an exception from this rule as due to fees, the balance can decrease.

4. **No change to other's balance ✔** (rule: noChangeToOthersBalances)
   By performing an operation, there is no change to other's assets that are not part of the arguments (to, from, tos, ...)

5. **Valid decrease to BalanceOf ✔** (rule: validDecreaseToBalanceOf)

   withdraw, transfer, and transferMultiple are the only operations that may decrease the balance of a user.

   ```
   { v = toAmount(token, balanceOf[token][a]) }

         op

   { toAmount(token, balanceOf[token][a]) < v ⇒

   (op = withdraw() ∨ op = transfer() ∨ op = transferMultiple()) }
   ```

6. **Additivity of withdraw of shares ✔**

   Withdraw of shares is additive, i.e., it doesn't matter whether some amount of shares is performed in one larger transaction or in two smaller ones.

   ```
   withdraw(token, from, to, 0, x); withdraw(token, from, to, 0, y)
   ~ withdraw(token, from, to, 0, x + y)
   ```

7. **Flashloan ✔** (rule: totalAssetsAfterFlashLoan)

   The total assets of the system do not decrease through a flash loan.

   ```
   { v = token.balanceOf(bentoBox) }

         flashloan()

   { token.balanceOf(bentoBox) ≥ v }
   ```

   We proved this for cases where flashloan may callback one of BentoBox's public functions.

8. **Strategy ✔** (rule: zeroStrategy)

   A zero strategy contract always has zero holding in its strategy.

   ```
   strategy(token) = 0 ⇒ getStrategyTokenBalance(token) == 0
   ```

Verification of a Strategy

A strategy is associated with a **token** and invests tokens on behalf of an **owner.** The contract **sushiStrategy** is verified against the following properties, in addition to a symbolicStrategy representing a general strategy.

Functions

1. **harvest(uint balance) : int**
   Transfers to the owner any positive profit made by the strategy on balance (from owner's view), and returns the amount of profit which can be negative.

   After a harvest operation, the strategyCurrentAssets must match the strategy balance passed in plus the returned difference. (The BentoBox will update the balance with the returned difference.)

2. **withdraw(uint amount, uint balance) : uint**
   Withdraws assets and transfers it to the owner. Returns the actual amount transferred, which can differ from amount due to rounding errors.

3. **exit(uint balance) : int**
   Withdraws all assets and transfers them to the owner. Returns the difference between balance (from the owner's view) and the actual amount transferred.

4. **strategyCurrentAssets() : uint**
   Returns the current assets in the strategy. This is an additional getter function for the purpose of verification.

Properties

1. **Integrity of harvest ✔**
   Transfers excess tokens above balance. On a positive profit, owner's balance increases and the return value is the profit above balance. On negative profit, the negative profit is returned.

```
{ v = token.balanceOf(owner) ∧ s = strategyCurrentAssets() }

      r = harvest(balance)

{ (s ≥ balance ⇒ r ≥ 0 ∧ token.balanceOf(owner) = v + r) ∧

s = balance + r }
```

2. **Integrity of withdraw ✔**

   A withdraw operation increases the withdrawer's balance by the withdrawn amount.

```
{ v = token.balanceOf(owner) }

      r = withdraw(amount, balance)

{ token.balanceOf(owner) = v + r }
```

3. **Integrity of Exit ✔**

   The exit operation transfers all of the strategy's assets to the owner.

```
{ v = token.balanceOf(owner) ⋀ s = strategyCurrentAssets() }

      r = exit(balance)

{ (token.balanceOf(owner) = v + balance + r) ⋀
                                        (s = balance + r) }
```