Formal Verification of ConstantProductPool

Summary

This document describes the specification and verification of ConstantProductPool by SushiSwap using the Certora Prover. The work was undertaken while the contract was in development and concluded on Nov 4th, 2021.

The scope of our verification was ConstantProductPool contract, a liquidity pool that uses a constant product market maker and consists of two tokens. The LP tokens are called Trident tokens.

The Certora Prover proved the implementation of ConstantProductPool is correct with respect to the formal rules written by the Certora and SushiSwap team.

The next section formally defines high level specifications of ConstantProductPool. The results of running the Certora Prover are available at:

- ConstantProductPool
- IntegrityOfTotalSupply
- noChangeToOtherBalances
- Sanity and Reenctrancy
- Blog-Post detailing the critical issue below

List of Main Issues Discovered

Severity: Critical

Issue:	Loss of all assets
Rules Broken:	integrityOfTotalSupply

Issue:	Loss of all assets
Description:	A burnSingle opertaion can empty a token's reserve. This can be leveraged to an exploit draining both pool's reserves, using a burnSingle followed by a swap.
Mitigation/Fix:	Updated the burnSingle method to compute burn amount of out reserve and not balance.

Severity: High

Issue:	Wrong amount out for burnSingle
Rules Broken:	burnTokenAdditivity
Description:	After burning the Trident tokens, the burnSingle method would always swapping token0 for token1 irrespective of the tokenOut.
Mitigation/Fix:	Updated the burnSingle method to swap the correct tokens based on the tokenOut.

Severity: High

Issue:	Non-parametric burnSingle
Rules Broken:	pathSanityForToken0
Description:	Users could only burn token1 using the burnSingle method because of a faulty require statement.
Mitigation/Fix:	Updated the burnSingle method to be able to burn both tokens based on the tokenOut.

Severity: Medium

Issue:	Denial of Service
Rules Broken:	integrityOfTotalSupply (on older version of the code)

Issue:	Denial of Service
Description:	If either of the reserves become non-zero when Trident's total supply is zero, then the users would not be able to mint anymore. Furthermore, the _update method was not checking for division by zero when reserve0 is zero, which could cause reverts and halt the system.
Mitigation/Fix:	All operations are possible only on a non-empty pool. `_update` method fixed to check for division by zero.

Severity: Medium

Issue:	Loss of assets
Rules Broken:	afterOpBalanceEqualsReserve
Description:	The unit for pool's balances and reserves is BentoBox shares, but at some places, they were confused as amounts and converted to shares before transferring.
Mitigation/Fix:	Removed the unnecessary conversions from amounts to shares in ConstantProductPool.

Severity: low

Issue:	Validity of pool's tokens
Rules Broken:	validityOfTokens & tokensNotTrident
Description:	Pool's tokens could be the zero address or the ConstantProductPool contract itself.
Mitigation/Fix:	Updated the constructor to check the validity of token addresses.

Severity: low

Issue:	Invalid pool
Rules Broken:	integrityOfTotalSupply (on older version of the code

Issue:	Invalid pool
Description:	When there is a migrator, there is no check that the balance now is greater than zero.
Mitigation/Fix:	This feature has been removed

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 the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit 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

✔️ indicates the rule is formally verified on the latest reviewed commit. We write ✔️* when the rule was verified under some simplifying assumption and/or when the verification run timeouts for some functionality of the code (and specify the functions below).

 indicates the rule was violated under one of the tested versions of the code.

Verification of ConstantProductPool

ConstantProductPool is one of the contracts from the Trident system by SushiSwap. The contract works with SushiSwap's BentoBox and TridentRouter, which is another contract of the Trident system. It is a liquidity pool that uses a constant product market maker and consists of two tokens, token0 and token1. Users can deposit liquidity in terms of token0 and token1 to mint Trident tokens. The contract also provides the functionality of swapping tokens and burning Trident tokens to get back liquidity in terms of token0 and token1 or either one of them.

The pool keeps track of the BentoBox balances for both tokens in the form of reserves. Whenever these balances change and an operation is performed, the reserves are updated accordingly. The pool is in a balanced state when the BentoBox balances for both tokens equal the internal state variables keeping track of the reserves.

Assumptions

- Verified using a simplified version of BentoBox and square root function.
- Loops are unwinded 4 times.
- Token1 is not the zero address (a safe assumption guaranteed by the ConstantProductPoolFactory).
- For certain operations, it is assumed that the recipient is not the ConstantProductPool.
- TridentERC20 has no overflows.
- `msg.sender` is not the ConstantProductPool.
- Simplified Trident callbacks.

Properties

1. Validity of pool tokens ✔️

- Tokens cannot be the zero address or equal to each other.
- Tokens cannot be the Trident token.

2. Pool's reserves are always less than or equal to pool's balances ✔️

3. Integrity of Trident's total supply ✔️*

- Trident's `totalSupply` is zero if and only if both the reserves are zero * BurnSingle and FlashSwap timed out. Burn was verified under the assumption that `kLast=0`, meaning that the Bar Fee was ignored.

4. Operations are parametric on tokens ✔️

- Every operation that takes a token as a parameter should support both tokens.

5. No operation changes a balanced pool's balances ✔️

6. After every operation, a pool's balances and reserves are equal ✔️

7. Minting Trident is not possible for balanced pools ✔️

8. No operation may change some other user's assets ✔️

- In case the recipient is the other user, their assets can increase.

11. Constant Product Curve is increasing for swaps ✔️

- The product of reserves before a swap is always less than or equal to the product of reserves after the swap.

12. Characteristics of `getAmountOut` ✔️

- If `amountIn` is zero, `amountOut` should always be zero.
- If `amountIn` token's reserve is zero, amountOut should always be zero.
- `amountOut` cannot be greater than the output token's reserve.

13. Minting zero Trident liquidity is not possible ✔️

14. No reentrancy for important functions ✔️

15. Integrity of token's BentoBox balances ✔️*

- If the bentoBox balance of one token decreases then the other token’s BentoBox should increase or Trident's `totalSupply` should decrease. * FlashSwap timed out. BurnSingle and Burn were verified under the assumption that kLast=0, meaning that the Bar Fee was ignored.