Balancer V2

Security Assessment

April 5, 2021

Prepared For:
Mike McDonald | *Balancer Protocol*
mike@balancer.finance

Fernando Martinelli | *Balancer Protocol*
fernando@balancer.finance

Prepared By:
Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Alexander Remie | *Trail of Bits*
alexander.remie@trailofbits.com

Executive Summary

From March 22 to April 2, 2021, Balancer engaged Trail of Bits to review the security of Balancer V2. Trail of Bits conducted this assessment over four person-weeks, with two engineers working from commits `2c84113` and `bce652f` from the balancer-core-v2 repository.

Trail of Bits performed a previous audit of this project over six person-weeks in January 2021. During that audit, 17 issues were found, including 7 related to rounding. Appendix H contains our rounding recommendations resulting from this initial audit.

In the first week of this audit, we focused on understanding the codebase, including the components' interactions with each other, and looked for the most common Solidity security flaws. We concentrated on the vault. In the second week, we focused on the pool contracts and fees system. We also used Echidna to check for rounding errors in the math contracts used by the pools, with a focus on the stable pool.

We found 14 issues. One of the high-severity issues (TOB-BALANCER-001) could allow an asset manager to drain a pool of all of the tokens not under his or her management. We also found many arithmetic issues in the stable pool, including some that could lead to token draining.

In addition to the security findings, we identified code quality issues not related to any particular vulnerability, which are discussed in Appendix C. Appendix D contains recommendations on evaluating arbitrary ERC20 tokens, and Appendix E provides recommendations on arbitrary pools. Appendix F contains a script built on top of Slither to review the contracts' `nonReentrant` modifiers. An Echidna-based differential fuzzer of the Curve and Balancer stable pool arithmetic is provided in Appendix G.

Overall, the Balancer V2 project follows best practices. The code's structure has been improved since our original audit, and Balancer has avoided the main Solidity pitfalls. However, the code's underlying complexity and several moving parts increase the likelihood of mistakes. Balancer implemented our rounding recommendations for the `WeightedPool`, but the rounding issues found in the `StablePool` indicate that the code's arithmetic has room for improvement and that more issues might be present. Finally, at the time of the audit, the code was unstable and undergoing changes, including some that were not reviewed in this audit.

Trail of Bits recommends that Balancer take the following steps:

- Address all reported issues.
- Implement additional Echidna tests to validate the rounding of all arithmetic.

- Document the dangers of using non-standard ERC20 tokens (e.g., deflationary tokens), and, using Appendix D, develop an ERC20 token checklist for users.
- Document the pools’ assumptions and develop a user checklist for arbitrary pools based on Appendix E.
- Perform an audit once the codebase has stabilized.

Project Dashboard

Application Summary

Name	Balancer V2
Version	`2c84113`, `bce652f`
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	March 22 – April 2, 2021
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■■
Total Medium-Severity Issues	3	■■■
Total Low-Severity Issues	5	■■■■■
Total Informational-Severity Issues	3	■■■
Total Undetermined-Severity Issues	1	■
Total	14	

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Data Validation	8	■■■■■■■■
Undefined Behavior	4	■■■■
Total	14	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory**. The project uses a robust authentication and authorization system.
Arithmetic	**Moderate**. Many issues stem from a lack of proper rounding validation, and more of those issues might be present. The codebase would benefit from case-by-case handling of the rounding direction.
Assembly Use	**Moderate**. While the use of assembly is limited and does not lead to any security issues, it could be reduced by sharing structures across contracts.
Centralization	**Weak.** There are limits on privileged users' actions. However, an asset manager could easily abuse his or her privileges to drain a pool of all other assets (TOB-BALANCER-001). Additionally, an admin could toggle the emergency period between active and inactive to selectively block transactions (TOB-BALANCER-014).
Code Stability	**Weak.** The code was undergoing significant changes during the audit and will likely continue to evolve before reaching its final version.
Upgradeability	**Not Applicable.** The system cannot be upgraded.
Function Composition	**Satisfactory.** The overall code is well structured, and most of the functions are small and have clear purposes. The critical functions can be easily extracted for testing.
Front-Running	**Moderate**. Most functions contain arguments with limits to prevent sudden changes that could negatively impact users' expectations and ability to execute transactions as intended. However, a privileged user could front-run withdrawals to quickly update the withdrawal fee (TOB-BALANCER-011).
Monitoring	**Satisfactory.** Most functions emit events. However, one function for changing the swap fee does not (TOB-BALANCER-004). Additionally, the emission of a transfer event upon token burning could confuse users monitoring the events of the system (TOB-BALANCER-002).
Specification	**Moderate.** The constant product arithmetic is well documented on https://balancer.finance/. However, the overall system would benefit from cleaner documentation on this architecture and assumption.

	Revising the documentation on fees and the components' interactions would also improve code readability.
Testing & Verification	**Moderate.** The system has suitable unit tests but would benefit from the addition of fuzzing or formal methods to ensure that the arithmetic meets expectations. Many issues in this report were found using Echidna.

Engagement Goals

The engagement was scoped to provide a security assessment of the Balancer V2 smart contracts.

Specifically, we sought to answer the following questions:

- Is it possible to cause a DoS in a pool?
- Is it possible for asset managers or admins to abuse their abilities?
- Is it possible for pools to access funds belonging to other pools?
- Is the use of native and wrapped ETH implemented correctly?
- Do the arithmetic libraries correctly apply rounding?
- Are fees applied correctly?
- Can swaps be used to steal funds?
- Is the flash loan feature implemented correctly?
- Can the authentication/authorization scheme be circumvented?
- Is the internal balance accounting system working as expected?

Coverage

Authorizer + Authentication. Two small contracts that provide authorization and authentication features throughout the contracts. We manually reviewed the implementation to look for flaws that could allow an attacker to subvert the authentication/authorization features.

EmergencyPeriod. Implements an emergency period used in all of the pools and the vault. We manually reviewed the implementation to look for flaws that could allow the admin to turn the emergency period on and off at will.

AssetTransferHandler. Contains functions for sending and receiving ERC20 tokens and native/wrapped ETH. We manually reviewed the implementation for flaws related to wrapping/unwrapping ETH and sending/receiving ERC20 tokens.

FlashLoanProvider. Contains a single `flashLoan` function. We reviewed the implementation to identify any flaws that could enable an attacker to borrow tokens without repaying the flash loan.

InternalBalance. Stores the internal balance of each account as well as various helper functions to add to or withdraw from the internal balance. We reviewed the balance update functions to discern whether an attacker could use them to gain access to other users' funds. We also checked whether fees were correctly applied to the balance.

PoolRegistry. Contains the functions to register, join, or exit a pool and to register/deregister tokens of a pool, as well as helper functions to pack/unpack pool variables into/from a single `bytes32` variable. We reviewed the implementation of the pack/unpack functions. We also reviewed all pool configuration-related functions to find flaws that would enable a pool to access or overwrite another pool's data or to otherwise disturb the internal accounting of pool registrations. Additionally, we reviewed the application of the fees for joining and exiting a pool.

ProtocolFeesCollector, Fees. Contain functions to set global fee percentages and receives the collected fees, which can be withdrawn by the system admin. We manually checked that all functions correctly validated the input arguments and that the fee withdrawal process did not contain flaws.

Swaps. Contain the main functions to initiate swaps, including multiple swaps in one call. We checked that the (chained) swaps did not contain flaws that could allow an attacker to steal funds. We checked that the correct pool specialization functions were called in all places.

VaultAuthorization. Provides authorization helper functions and functions to set a relayer on behalf of an account. We checked that the authorization functions did not contain flaws and that relayers were used correctly.

Vault. Acts as the entry-point contract for Balancer and is where tokens are stored, user balances are updated, pools are registered, and flash loans are provided. We looked for flaws that could allow a malicious asset manager to steal assets. We checked the registration of pools, their tokens, and the addition and removal of liquidity for all three pool specialization types. We reviewed the use of native ETH and the conversion to wrapped ETH for flaws that could enable ETH theft. We reviewed the use of internal balances, including whether they could be used to steal tokens from the system. Additionally, we examined the storage of the packed balances in a bytes32 and the various extraction functions.

Pools (stable, weighted). We reviewed the pools' components, with a focus on their arithmetic and access controls. We confirmed that only the vault could call the join/exit operations. Using Echidna, we tested stable and weighted math and looked for flaws that could allow an attacker to gain tokens from swapping, joining, or exiting. Due to the time constraints, and because our original audit focused on the weighted pool, we concentrated our efforts on the stable pool arithmetic. Due to the number of stable pool arithmetic findings, we believe that more issues might be present.

Due to the time constraints, Trail of Bits could not explore the following areas:

- `BalancerHelpers.sol`
- `EnumerableMap.sol`
- `LogExpMath.sol` and its impact on callers

Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance coverage of certain areas of the contracts, including:

- Slither, a Solidity static analysis framework.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation.

Automated testing techniques augment our manual security review but do not replace it. Each method has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode; Echidna may not randomly generate an edge case that violates a property.

Automated Testing with Echidna

We implemented the following Echidna properties:

StableMath Arithmetic Properties

ID	Property	Result
1	`StableMath._calcOutGivenIn` cannot lead to free tokens.	FAILED (TOB-BALANCER-006)
2	`StableMath._calcOutGivenIn` cannot lead to free tokens with large balances.	PASSED
3	`StableMath._calcInGivenOut` cannot lead to free tokens.	FAILED (TOB-BALANCER-007)
4	`StableMath._calcInGivenOut` cannot lead to free tokens with large balances.	PASSED
5	`StableMath._calcTokenInGivenExactBptOut` cannot lead to free BPT tokens.	FAILED (TOB-BALANCER-008)
6	`StableMath._calculateInvariant` is monotonically increasing.	FAILED (TOB-BALANCER-014)

7	`StableMath._calculateInvariant` does not differ significantly from Curve's invariant.	FAILED (TOB-BALANCER-009)

WeightedMath Arithmetic Properties

ID	Property	Result
8	`WeightedMath._calcOutGivenIn` cannot lead to free tokens.	PASSED
9	`WeightedMath._calcOutGivenIn` cannot lead to free tokens with large balances.	PASSED
10	`WeightedMath._calcInGivenOut` cannot lead to free tokens.	PASSED
11	`WeightedMath._calcTokenInGivenExactBptOut` cannot lead to free BPT tokens.	PASSED
12	`WeightedMath._calculateInvariant` is monotonically increasing.	PASSED
13	The sum of all normalized weights in a pool is 1.	FAILED (TOB-BALANCER-012)

All the properties were run with pools containing four tokens, and with a TestLimit of five million.

Automated Testing with Slither

We implemented the following Slither property:

Property	Result
All `Vault` functions expected to have the `nonReentrant` modifier do have it.	PASSED (**APPENDIX F**)

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Document the fact that the manager has access to all of the funds in the pool, or allow one manager per pool, instead of one manager per token.** This will prevent user confusion and reduce the risks associated with malicious managers. TOB-BALANCER-001

❑ **Create a `Burn` event or prevent transfer to the zero address**. This will ensure that an attacker cannot confuse event monitors with incorrect burn events. TOB-BALANCER-002

❑ **Measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug.** The compiler optimizer was frequently the source of bugs and should be carefully reviewed. TOB-BALANCER-003

❑ **Add events for `BasePool.setSwapFee`, `Authorizer.changeAuthorizer`, and `Authorizer.changeRelayerAllowance`.** Events will facilitate contract monitoring and the detection of suspicious behavior. TOB-BALANCER-004

❑ **Add zero-value checks on all function arguments.** This will ensure that users can't accidentally set incorrect values, misconfiguring the system. TOB-BALANCER-005

❑ **Remove `FixedPoint.mul` and `FixedPoint.div`, and ensure that every function uses the appropriate rounding direction**. These functions are error-prone and can lead to rounding issues. TOB-BALANCER-006, TOB-BALANCER-007, TOB-BALANCER-008

❑ **Create two versions of `_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` that round up or down, respectively, and use them appropriately.** Several issues were caused by incorrect rounding strategies in these two functions. TOB-BALANCER-006, TOB-BALANCER-007, TOB-BALANCER-008

❑ **Investigate the differences between the Balancer and Curve invariants, and evaluate their impact on arbitrage opportunities and potential associated risks.** The impacts of their differences are unclear and should be reviewed. TOB-BALANCER-009

❑ **Disallow swap/join/exit operations if the balance of the `token in` has been emptied, or if the operations would empty the `token out`**. The weighted pool's method of handling a pool with an empty balance is fragile, and the existing mitigations could cease to work if the code is refactored. TOB-BALANCER-010

❑ **Add a minimum withdrawal requirement for withdrawal operations.** Otherwise, a malicious owner could front-run the withdrawal operations and make users pay higher-than-expected fees. TOB-BALANCER-011

❑ **Check that the sum of the normalized weights is equal to 10**18 in the weighted pool's constructor.** This will ensure that the pool's invariant is preserved at deployment. TOB-BALANCER-012

❑ **Apply the correct rounding in `StableMath._calculateInvariant`.** The current invariant is not monotonically increasing, which could lead to unexpected behavior in all of the stable pool's formulas. TOB-BALANCER-013

❑ **Document the emergency period-related abilities of an admin so that users will be well informed.** TOB-BALANCER-014

Long Term

❑ **Document the expectations for managers and their associated risks.** Consider providing smart contracts that can act as managers with on-chain restrictions. TOB-BALANCER-001

❑ **Document and test the expected behavior of all of the system's events.** Consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. TOB-BALANCER-002

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** The deployment requires optimizations, which are inherently risky. TOB-BALANCER-003

❑ **Use Slither.** Slither will catch many security issues while requiring a low effort. TOB-BALANCER-004, TOB-BALANCER-005

❑ **Use Echidna when designing a new pool to ensure that swap/join/exit operations and the pool's invariant are robust against rounding issues.** Many arithmetic issues were found using Echidna. TOB-BALANCER-006, TOB-BALANCER-007, TOB-BALANCER-008, TOB-BALANCER-010, TOB-BALANCER-012, TOB-BALANCER-013

❑ **Use differential fuzzing when designing a system for which another implementation exists.** Echidna can be used as a differential fuzzer to ensure that the implementation works as expected. TOB-BALANCER-009

❑ **Identify and document all parameters that can be changed by privileged users.** Ensure that updates to these parameters will have a limited impact on users' funds. TOB-BALANCER-011

❑ **Document the abilities of the privileged roles throughout the system.** That way, users will not be surprised when others exercise their privileges. TOB-BALANCER-014

Findings Summary

Per Balancer's request, the stable pool-related findings are located at the end of this section because of possible future deprecation.

#	Title	Type	Severity
1	Malicious manager can reinvest tokens to drain the pool	Undefined Behavior	High
2	Transfer to zero can lead to unexpected burns	Access Controls	Informational
3	Solidity compiler optimizations can be dangerous	Undefined Behavior	Informational
4	Missing events for critical operations	Auditing and Logging	Informational
5	Lack of zero check on functions	Data Validation	Low
10	Lack of robust safeguards for pools with an empty token in WeightedPool	Data Validation	High
11	Protocol fee front-run	Data Validation	Low
12	Sum of normalized weights can be different from 1	Data Validation	Low
14	Emergency period toggling can be used to selectively block transactions	Undefined Behavior	Low
6	`StableMath._calcOutGivenIn` may allow free swaps	Data Validation	Medium
7	`StableMath._calcInGivenOut` may allow free swaps	Data Validation	Medium
8	StableMath._calcTokenInGivenExactBptOut may allow an attacker to join for free	Data Validation	Medium
9	Balancer StablePool's invariant can differ significantly from Curve's invariant	Undefined Behavior	Undetermined
13	StablePool's invariant is not monotonically increasing	Data Validation	Low

1. Malicious manager can reinvest tokens to drain the pool

Severity: High
Type: Undefined Behavior
Target: `PoolRegistry.sol`

Difficulty: High
Finding ID: TOB-BALANCER-001

Description
Each token in a pool has a manager who can borrow tokens. A malicious manager could borrow the tokens, swap them for all the other tokens, and drain the pool.

`withdrawFromPoolBalance` allows a manager of a token in a pool to withdraw tokens:

```
function withdrawFromPoolBalance(bytes32 poolId, AssetManagerTransfer[] memory transfers)
    external
    override
    nonReentrant
    noEmergencyPeriod
{
    _ensureRegisteredPool(poolId);
    PoolSpecialization specialization = _getPoolSpecialization(poolId);

    for (uint256 i = 0; i < transfers.length; ++i) {
        IERC20 token = transfers[i].token;
        _ensurePoolAssetManagerIsSender(poolId, token);

        uint256 amount = transfers[i].amount;
        if (specialization == PoolSpecialization.MINIMAL_SWAP_INFO) {
            _minimalSwapInfoPoolCashToManaged(poolId, token, amount);
        } else if (specialization == PoolSpecialization.TWO_TOKEN) {
            _twoTokenPoolCashToManaged(poolId, token, amount);
        } else {
            _generalPoolCashToManaged(poolId, token, amount);
        }

        token.safeTransfer(msg.sender, amount);
        emit PoolBalanceChanged(poolId, msg.sender, token, -(amount.toInt256()));
    }
}
```

Figure 1.1: `vault/PoolRegistry.sol#L512-L537`.

It is expected that a manager can impact or steal only the tokens under his management. However, as a manager can reinvest the tokens directly and swap them for the other tokens in the pool, he can drain the pool entirely.

Exploit Scenario
Bob deploys a pool with four tokens, one of which Eve manages. Eve withdraws and swaps all the tokens she is managing for the others, effectively draining the pool.

Recommendations
Short term, take one of the following steps:

- Document the fact that the manager has access to all of the funds in the pool
- Allow one manager per pool, instead of one manager per token

This will prevent user confusion and reduce the risks associated with malicious managers.

Long term, document the expectations for managers and their associated risks. Consider providing smart contracts that can act as managers with on-chain restrictions.

2. Transfer to zero can lead to unexpected burns

Severity: Informational
Difficulty: Low
Type: Access Controls
Finding ID: TOB-BALANCER-002
Target: `BalancerPoolToken.sol`

Description
`BalancerPoolToken` uses a `Transfer`-to-zero event to indicate a burn. That same event can be triggered without burning tokens, which could cause confusion among off-chain monitors.

When tokens are burned, the total supply decreases, and a `Transfer`-to-zero event is emitted:

```
function _burnPoolTokens(address sender, uint256 amount) internal {
    uint256 currentBalance = _balance[sender];
    require(currentBalance >= amount, "INSUFFICIENT_BALANCE");

    _balance[sender] = currentBalance - amount;
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(sender, address(0), amount);
}
```

Figure 2.1: `pools/BalancerPoolToken.sol#L139-L146`

Users can emit the same event by calling `transfer` / `transferFrom` directly, but the total supply will not be changed. This divergence can be confusing to off-chain monitors and can make tracking the system's events more difficult.

Recommendations
Short term, either create a `Burn` event or prevent transfer to the zero address.

Long term, document and test the expected behavior of all of the system's events. Consider using a blockchain-monitoring system to track any suspicious behavior in the contracts.

3. Solidity compiler optimizations can be dangerous

Severity: Informational
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-BALANCER-003
Target: `hardhat.config.ts`

Description
In Balancer V2, optional compiler optimizations in Solidity are enabled.

There have been several optimization-related bugs in Solidity with security implications, and optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. It is therefore unclear how well they are being tested and exercised.

High-severity security issues caused by optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, a bug stemming from incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

There are likely latent bugs related to optimization and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Balancer contracts.

Recommendations
Short term, measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

4. Missing events for critical operations

Severity: Informational Difficulty: Low
Type: Auditing and Logging Finding ID: TOB-BALANCER-004
Target: `pools/BasePool.sol, vault/VaultAuthorization.sol`

Description
Several critical operations do not trigger events, which will make it difficult to check the behavior of the contracts once they have been deployed.

Ideally, the following critical operations should trigger events:

- `BasePool.setSwapFee`

```
    function setSwapFee(uint256 swapFee) external authenticate {
        require(swapFee <= _MAX_SWAP_FEE, "MAX_SWAP_FEE");
        _swapFee = swapFee;
    }
```

Figure 4.1: `pools/BasePool.sol#L181-L184`

- `VaultAuthorization.changeAuthorizer`

```
    function changeAuthorizer(IAuthorizer newAuthorizer) external
override nonReentrant authenticate {
        _authorizer = newAuthorizer;
    }
```

Figure 4.2: `vault/VaultAuthorization.sol#L42-L44`

- `VaultAuthorization.changeRelayerAllowance`

```
    function changeRelayerAllowance(address relayer, bool allowed)
external override nonReentrant noEmergencyPeriod {
        _allowedRelayers[msg.sender][relayer] = allowed;
    }
```

Figure 4.3: `vault/VaultAuthorization.sol#L50-L52`

Without events, users and blockchain-monitoring systems will not be able to easily detect suspicious behavior.

Recommendations
Short term, add events for `BasePool.setSwapFee`, `Authorizer.changeAuthorizer`, and `Authorizer.changeRelayerAllowance`. Events will facilitate contract monitoring and the detection of suspicious behavior.

Long term, add Slither, which found this issue, to your CI.

5. Lack of zero check on functions

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-BALANCER-005
Target: `pools/BasePoolFactory.sol`

Description
Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

These include the `constructor` in `BasePoolFactory`, which sets the `vault`. This contract serves as the base contract for the `StablePoolFactory` and `WeightedPoolFactory` contracts. If the `vault` is set to address zero, subsequent calls to create a `StablePool` or `WeightedPool` will revert; to set a correct `vault`, the pool factory will need to be redeployed.

```
    constructor(IVault _vault) {
        vault = _vault;
    }
```

Figure 5.3: `pools/BasePoolFactory.sol#L26-L28`

This issue is also prevalent in the following functions:

- `AssetTransferHandler.constructor`
- `Authorizer.constructor`

Exploit Scenario
Alice deploys a `StablePoolFactory` and accidentally sets the `vault` to `address(0)`. When she invokes the `create` function to create a `StablePool`, the transaction reverts because the `vault` has been set to `address(0)`.

Recommendations
Short term, add zero-value checks on all function arguments to ensure that users can't accidentally set incorrect values, misconfiguring the system.

Long term, use Slither, which will catch functions that do not have zero checks.

10. Lack of robust safeguards for pools with an empty token in `WeightedPool`

Severity: High
Type: Data Validation
Target: *WeightedMath.sol, WeightedPool.sol*

Difficulty: High
Finding ID: TOB-BALANCER-010

Description

The `WeightedPool` has several corner cases that could either enable a user to receive tokens for free or trap the liquidity if one of the pools has an empty balance. It might not be possible for a user to empty a pool's underlying token balance; however, we recommend adding safeguards to improve the pools' robustness.

Cases that could lead to free tokens

`_calcTokenInGivenExactBptOut` determines the number of tokens that a user must send to receive a given number of the tokens in a pool (i.e., single asset deposit):

```
function _calcTokenInGivenExactBptOut(
    uint256 tokenBalance,
    uint256 tokenNormalizedWeight,
    uint256 bptAmountOut,
    uint256 bptTotalSupply,
    uint256 swapFee
) internal pure returns (uint256) {
    /*********************************************************************************
    // tokenInForExactBPTOut                                                        //
    // a = tokenAmountIn                                                            //
    // b = tokenBalance                 /  /  totalBPT + bptOut    \    (1 / w)   \   //
    // bptOut = bptAmountOut    a = b * | | --------------------- | ^        - 1  |   //
    // bpt = totalBPT                   \  \     totalBPT         /              /   //
    // w = tokenWeight                                                              //
    *********************************************************************************/

    // Token in, so we round up overall.

    // Calculate the factor by which the invariant will increase after minting BPTAmountOut
    uint256 invariantRatio = bptTotalSupply.add(bptAmountOut).divUp(bptTotalSupply);

    // Calculate by how much the token balance has to increase to cause invariantRatio
    uint256 tokenBalanceRatio = FixedPoint.powUp(invariantRatio,
FixedPoint.ONE.divUp(tokenNormalizedWeight));
    uint256 tokenBalancePercentageExcess = tokenNormalizedWeight.complement();
    uint256 amountInAfterFee = tokenBalance.mulUp(tokenBalanceRatio.sub(FixedPoint.ONE));

    uint256 swapFeeExcess = swapFee.mulUp(tokenBalancePercentageExcess);

    return amountInAfterFee.divUp(swapFeeExcess.complement());
}
```

Figure 10.1: `pools/weighted/WeightedMath.sol#L172-L201`

If the balance of the `tokenIn` is zero, the `amountIn` required will be zero. An attacker may be able to swap tokens for free if one of the tokens has a balance of zero. Similar issues can occur in other functions, such as `WeightedMath._calcInGivenOut.`

Cases that could lead to a trapped pool

When a user wants to join or exit the pool, `_getDueProtocolFeeAmounts` is called.

```
uint256[] memory dueProtocolFeeAmounts = _getDueProtocolFeeAmounts(
```

Figure 10.1: `pools/weighted/WeightedPool.sol#L266`

```
dueProtocolFeeAmounts = _getDueProtocolFeeAmounts(
```

Figure 10.2: `pools/weighted/WeightedPool.sol#L375`

`_getDueProtocolFeeAmounts` divides by `currentInvariant`.

```
uint256 invariantRatio = previousInvariant.divUp(currentInvariant);
```

Figure 10.3: `pools/weighted/WeightedPool.sol#L500`

If one of the balances is zero, the pool's invariant will be zero.

```
function _calculateInvariant(uint256[] memory normalizedWeights, uint256[] memory balances)
    internal
    pure
    returns (uint256 invariant)
{
    /**********************************************************************************
    // invariant                _____                                                //
    // wi = weight index i      | |      wi                                          //
    // bi = balance index i     | |  bi ^   = i                                      //
    // i = invariant                                                                 //
    **********************************************************************************/
    InputHelpers.ensureInputLengthMatch(normalizedWeights.length, balances.length);

    invariant = FixedPoint.ONE;
    for (uint8 i = 0; i < normalizedWeights.length; i++) {
        invariant = invariant.mul(FixedPoint.pow(balances[i], normalizedWeights[i]));
    }
}
```

Figure 10.4: `pools/weighted/WeightedMath.sol#L37-L54`

If one of the balances is zero, the balance will be divided by zero, causing join/exit operations to revert.

Due to the current rounding strategies, we were not able to empty a pool through swap/exit operations. However, a code refactoring or a specific edge case might allow an attacker to use the operations to turn a profit.

Exploit Scenario
Bob deploys a pool with four tokens. Eve finds a way to empty the pool of one of the tokens, consequently draining it of the other three.

Recommendations
Short term, disallow swap/join/exit operations if the `token in` has an empty balance or if the operations would empty it out.

Long term, use Echidna when designing a new pool to ensure that the swap/join/exit operations are robust against balance-emptying attempts and similar edge cases.

11. Protocol fee front-run

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-BALANCER-011
Target: *PoolRegistry.sol, InternalBalance.sol, ProtocolFeesCollector.sol*

Description
Privileged users can front-run withdrawal operations to increase withdrawal fees, reducing the amount of assets other users will receive when they execute withdrawals.

Withdrawal operations carry a fee:

```
uint256 withdrawFee = toInternalBalance ? 0 :_calculateProtocolWithdrawFeeAmount(amountOut);
_sendAsset(assets[i], amountOut.sub(withdrawFee), recipient, toInternalBalance, false);
```

Figure 11.1: vault/PoolRegistry.sol#L388-L389

```
if (taxableAmount > 0) {
     uint256 feeAmount = _calculateProtocolWithdrawFeeAmount(taxableAmount);
    _payFee(token, feeAmount);
    amountToSend = amountToSend.sub(feeAmount);
}

_sendAsset(asset, amountToSend, recipient, false, false);
```

Figure 11.2: vault/InternalBalance.sol#L99-L105

The fee is set through `setProtocolWithdrawFee`:

```
function setWithdrawFee(uint256 newWithdrawFee) external authenticate {
    require(newWithdrawFee <= _MAX_PROTOCOL_WITHDRAW_FEE, "WITHDRAW_FEE_TOO_HIGH");
    _withdrawFee = newWithdrawFee;
    emit WithdrawFeeChanged(newWithdrawFee);
}
```

Figure 11.3: vault/ProtocolFeesCollector.sol#L80-L84

Withdrawal operations do not specify the amount of assets a user should receive after the fee has been applied. As a result, if a user sends a withdrawal transaction and the fee is updated before the withdrawal has been minted, the user will receive a lower-than-expected amount of assets.

Exploit Scenario
Eve sets the fee to 0. Bob withdraws assets worth $10,000,000. Eve front-runs the withdrawal and sets the fee to 0.5%. As a result, Bob loses $50,000.

Recommendations

Short term, add a minimum withdrawal requirement for withdrawal operations.

Long term, identify and document all parameters that can be changed by privileged users. Ensure that updates to these parameters will have a limited impact on users’ funds.

12. Sum of normalized weights can be different from 1

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-BALANCER-012
Target: *`WeightedPool.sol`*

Description

The weighted pool token normalizes token weights such that a weight is a percentage of the tokens' total weight. The computation expects the sum of all weights to be equal to 1, but rounding may result in a different sum.

To compute normalized weights, the weighted pool constructor sums the weights and calculates each weight as the weight divided by the sum of the weights:

```
// Check valid weights and compute normalized weights
uint256 sumWeights = 0;
for (uint8 i = 0; i < weights.length; i++) {
    sumWeights = sumWeights.add(weights[i]);
}

uint256 maxWeightTokenIndex = 0;
uint256 maxNormalizedWeight = 0;
uint256[] memory normalizedWeights = new uint256[](weights.length);

for (uint8 i = 0; i < normalizedWeights.length; i++) {
    uint256 normalizedWeight = weights[i].div(sumWeights);
    require(normalizedWeight >= _MIN_WEIGHT, "MIN_WEIGHT");
    normalizedWeights[i] = normalizedWeight;

    if (normalizedWeight > maxNormalizedWeight) {
        maxWeightTokenIndex = i;
        maxNormalizedWeight = normalizedWeight;
    }
}
```

Figure 12.1: `pools/weighted/WeightedPool.sol#L72-L91`

Due to the arithmetic precision loss in `weights[i].div(sumWeights)`, the sum of the normalized weights can be slightly higher or lower than 1. This may break the underlying assumptions of the pool.

Exploit Scenario

Bob deploys a pool with 4 tokens. Their weights are 1, 1, 1, and 3. The sum of the normalized weights is equal to 10**18 + 1, which breaks one of the pool's invariants.

Recommendations

Short term, check that the sum of the normalized weights is equal to 10**18 in the weighted pool's constructor. This will ensure that the pool's invariant is preserved at deployment.

Long term, identify the system's invariants, and use Echidna to check their robustness.

14. Emergency period toggling can be used to selectively block transactions

Severity: Low
Type: Undefined Behavior
Target: *EmergencyPeriod.sol*

Difficulty: High
Finding ID: TOB-BALANCER-014

Description
The `EmergencyPeriod` contract allows the admin to repeatedly toggle the emergency period between active and inactive until the `_emergencyPeriodEndDate` has passed. As such, a malicious admin could cause transactions of the admin's choosing to fail by front-running them and then quickly enabling the emergency period.

```
    function _setEmergencyPeriod(bool active) internal {
        require(block.timestamp < _emergencyPeriodEndDate, "EMERGENCY_PERIOD_FINISHED");
        _emergencyPeriodActive = active;
        emit EmergencyPeriodChanged(active);
    }
```

Figure 14.1: `lib/helpers/EmergencyPeriod.sol#L53-L57`

```
    function setEmergencyPeriod(bool active) external authenticate {
        _setEmergencyPeriod(active);
    }
```

Figure 14.2: `pools/BasePool.sol#L186-L188`

```
    function setEmergencyPeriod(bool active) external authenticate {
        _setEmergencyPeriod(active);
    }
```

Figure 14.3: `vault/Vault.sol#L69-L71`

Exploit Scenario
Eve becomes an admin before the emergency period has ended. Eve monitors the mempool for transactions executed by Bob. When she finds one, she front-runs the transaction, quickly enables the emergency period, and then deactivates that period.

Recommendations
Short term, document the emergency period-related abilities of an admin so that users will be well informed.

Long term, clearly document the abilities of the privileged roles throughout the system. That way, users will not be surprised when others exercise their privileges.

6. `StableMath._calcOutGivenIn` may allow free swaps

Severity: Medium
Type: Data Validation
Target: *`StableMath.sol`*

Difficulty: High
Finding ID: TOB-BALANCER-006

Description
The out-given-in function has rounding to calculate swaps, which could enable an attacker to obtain tokens at no cost.

`StableMath._calcOutGivenIn` computes the number of tokens that one user will receive based on the number of tokens sent:

```
function _calcOutGivenIn(
    uint256 amplificationParameter,
    uint256[] memory balances,
    uint256 tokenIndexIn,
    uint256 tokenIndexOut,
    uint256 tokenAmountIn
) internal pure returns (uint256) {
    /**************************************************************************************
    // outGivenIn token x for y - polynomial equation to solve                           //
    // ay = amount out to calculate                                                      //
    // by = balance token out                                                            //
    // y = by - ay (finalBalanceOut)                                                     //
    // D = invariant                                  D                     D^(n+1)       //
    // A = amplification coefficient    y^2 + (S - --------- - D) * y - ------------ = 0 //
    // n = number of tokens                         (A * n^n)             A * n^2n * P    //
    // S = sum of final balances but y                                                   //
    // P = product of final balances but y                                               //
    **************************************************************************************/

    // Amount out, so we round down overall.

    uint256 invariant = _calculateInvariant(amplificationParameter, balances);

    balances[tokenIndexIn] = balances[tokenIndexIn].add(tokenAmountIn);

    uint256 finalBalanceOut = _getTokenBalanceGivenInvariantAndAllOtherBalances(
        amplificationParameter,
        balances,
        invariant,
        tokenIndexOut
    );

    balances[tokenIndexIn] = balances[tokenIndexIn].sub(tokenAmountIn);
    return balances[tokenIndexOut].sub(finalBalanceOut).sub(1);
}
```

Figure 6.1: `pools/stable/StableMath.sol#L89-L124`

`_calcOutGivenIn` performs:

$$invariant = calculateInvariant(amp, balances)$$
$$balances[tokenIn] = balances[tokenIn] + tokenAmountIn$$
$$finalBalanceOut = getTokenBalance[..]Balances(amp, balances, newInvariant, tokenIndex)$$

`_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` contain operations that can round up or down according to their values.

For example, `_getTokenBalanceGivenInvariantAndAllOtherBalances` uses `FixedPoint.mul`, which can round up or down based on the value:

```
tokenBalance = tokenBalance.mul(tokenBalance)
                           .add(c)
                           .divUp(
                               Math.mul(tokenBalance, 2)
                                   .add(b)
                                   .sub(invariant)
                           );
```

Figure 6.2: `pools/stable/StableMath.sol#L489`

As a result, `_calcOutGivenIn` can round in a direction that allows an attacker to receive free tokens by either swapping with 0 `token in` for a small amount of `token out` or accumulating dust through calls to `StableMath._calcOutGivenIn.`

We classified this issue as one of medium severity, as we were not able to generate free tokens with large balances. However, that might still be possible in certain edge cases.

Exploit Scenario

- Amp is `4168774334271564283904`.
- The pool has 4 tokens, and the balances are `35303029`, `1524785991`, `323536`, and `323534`.
- Eve swaps 62,439,391 of the tokens in index 1 for 0 tokens in index 0.

Recommendations

Short Term

- Remove `FixedPoint.mul` and `FixedPoint.div`, and ensure that every function uses the appropriate rounding direction.
- Create two versions of `_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` that round up or down, respectively, and use them appropriately.

Long term, use Echidna when designing a new pool to ensure that the swap/join/exit operations are robust against rounding issues.

7. `StableMath._calcInGivenOut` may allow free swaps

Severity: Medium
Type: Data Validation
Target: *`StableMath.sol`*

Difficulty: High
Finding ID: TOB-BALANCER-007

Description
The in-given-out function has rounding to calculate swaps, which could enable an attacker to obtain tokens at no cost.

`StableMath._calcOutGivenOut` computes the number of tokens that one user will receive based on the number of tokens sent:

```
function _calcInGivenOut(
    uint256 amplificationParameter,
    uint256[] memory balances,
    uint256 tokenIndexIn,
    uint256 tokenIndexOut,
    uint256 tokenAmountOut
) internal pure returns (uint256) {

    /**************************************************************************************
    // inGivenOut token x for y - polynomial equation to solve                           //
    // ax = amount in to calculate                                                       //
    // bx = balance token in                                                             //
    // x = bx + ax (finalBalanceIn)                                                      //
    // D = invariant                                   D                   D^(n+1)       //
    // A = amplification coefficient    x^2 + (S - --------- - D) * x - ------------ = 0 //
    // n = number of tokens                         (A * n^n)           A * n^2n * P     //
    // S = sum of final balances but x                                                   //
    // P = product of final balances but x                                               //
    **************************************************************************************/

    // Amount in, so we round up overall.

    uint256 invariant = _calculateInvariant(amplificationParameter, balances);

    balances[tokenIndexOut] = balances[tokenIndexOut].sub(tokenAmountOut);

    uint256 finalBalanceIn = _getTokenBalanceGivenInvariantAndAllOtherBalances(
        amplificationParameter,
        balances,
        invariant,
        tokenIndexIn
    );

    balances[tokenIndexOut] = balances[tokenIndexOut].add(tokenAmountOut);
    return finalBalanceIn.sub(balances[tokenIndexIn]).add(1);
}
```
Figure 7.1: `pools/stable/StableMath.sol#L129-L164`

`_calcOutGivenOut` performs:

$$invariant = calculateInvariant(amp, balances)$$
$$balances[tokenOut] = balances[tokenOut] - tokenAmountOut$$
$$finalBalanceIn = getTokenBalance[..]Balances(amp, balances, newInvariant, tokenIndex)$$

`_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` contain operations that can round up or down according to their values.

For example, `_getTokenBalanceGivenInvariantAndAllOtherBalances` uses `FixedPoint.mul`, which can round up or down based on the value:

```
tokenBalance = tokenBalance.mul(tokenBalance)
                           .add(c)
                           .divUp(
                               Math.mul(tokenBalance, 2)
                                   .add(b)
                                   .sub(invariant)
                           );
```
Figure 7.2: `pools/stable/StableMath.sol#L489`

As a result, `_calcOutGivenOut` can round in a direction that allows an attacker to receive free tokens by accumulating dust with calls to `StableMath._calcOutGivenOut`.

We classified this issue as one of medium severity, as we were not able to generate free tokens with large balances. However, that might still be possible in certain edge cases.

Exploit Scenario

- Amp is 10**18.
- The balances are 20,369, 17,465,037,809, and 1.
- Eve swaps 1 wei of the tokens in index 0 and receives 37,808 wei of the tokens in index 2.
- Eve then swaps 37,808 wei of the tokens in index 2 and receives 20,369 wei of the tokens in index 0.
- As a result, Eve receives 20,368 wei of the tokens in index 0 for free.

Recommendations

Short Term

- Remove `FixedPoint.mul` and `FixedPoint.div`, and ensure that every function uses the appropriate rounding direction.

- Create two versions of `_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` that round up or down, respectively, and use them appropriately.

Long term, use Echidna when designing a new pool to ensure that the swap/join/exit operations are robust against rounding issues.

8. StableMath._calcTokenInGivenExactBptOut may allow an attacker to join for free

Severity: Medium
Type: Data Validation
Target: *StableMath.sol*

Difficulty: High
Finding ID: TOB-BALANCER-008

Description

The "join" operation for token-in-for-exact-bpt-out has rounding, which could allow an attacker to join a pool without paying any tokens.

`StableMath._calcTokenInGivenExactBptOut` determines the number of tokens that a user must send to receive a given number of Bpt tokens (for a single asset):

```
function _calcTokenInGivenExactBptOut(
    uint256 amp,
    uint256[] memory balances,
    uint256 tokenIndex,
    uint256 bptAmountOut,
    uint256 bptTotalSupply,
    uint256 swapFee
) internal pure returns (uint256) {
    // Token in, so we round up overall.

    // Get current invariant
    uint256 currentInvariant = _calculateInvariant(amp, balances);

    // Calculate new invariant
    uint256 newInvariant =
bptTotalSupply.add(bptAmountOut).divUp(bptTotalSupply).mulUp(currentInvariant);

    // First calculate the sum of all token balances which will be used to calculate
    // the current weight of token
    uint256 sumBalances = 0;
    for (uint256 i = 0; i < balances.length; i++) {
        sumBalances = sumBalances.add(balances[i]);
    }

    // get amountInAfterFee
    uint256 newBalanceTokenIndex = _getTokenBalanceGivenInvariantAndAllOtherBalances(
        amp,
        balances,
        newInvariant,
        tokenIndex
    );
    uint256 amountInAfterFee = newBalanceTokenIndex.sub(balances[tokenIndex]);

    // Get tokenBalancePercentageExcess
    uint256 currentWeight = balances[tokenIndex].divDown(sumBalances);
    uint256 tokenBalancePercentageExcess = currentWeight.complement();

    uint256 swapFeeExcess = swapFee.mulUp(tokenBalancePercentageExcess);

    return amountInAfterFee.divUp(swapFeeExcess.complement());
}
```
Figure 8.1: `pools/stable/StableMath.sol#L129-L164`

`_calcTokenInGivenExactBptOut` performs:

$$currentInvariant \;=\; calculateInvariant(amp,\; balances)$$

$$newInvariant \;=\; \frac{BPTTotalSupply + BPTAmountOut}{BPTTotalSupply}\nearrow *_{\nearrow} currentInvariant$$

$$sumBalances \;=\; \sum_i balances[i]$$

$$newBalanceTokenIndex \;=\; getTokenBalance[..]Balances(amp,\; balances,\; newInvariant,\; tokenIndex)$$

`_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` contain operations that can round up or down according to their values.

For example, `_getTokenBalanceGivenInvariantAndAllOtherBalances` uses `FixedPoint.mul`, which can round up or down based on the value.

```
tokenBalance = tokenBalance.mul(tokenBalance)
                           .add(c)
                           .divUp(
                               Math.mul(tokenBalance, 2)
                                   .add(b)
                                   .sub(invariant)
                           );
```
Figure 8.2: `pools/stable/StableMath.sol#L489`

As a result, `_calcTokenInGivenExactBptOut` can round in a direction that allows an attacker to receive Bpt tokens without paying any tokens.

We classified this issue as one of medium severity, as we were not able to generate free tokens with large balances. However, that might still be possible in certain edge cases.

Exploit Scenario

- Amp is `2875840079131296399935.`
- There are 4 tokens in the pool, each of which has a balance of 100,000,000,000,000,000.
- The current Bpt supply is 5,690,710,977,914,755,780,762,205.

- Eve joins the pool to obtain 1,000,021 Bpt tokens without having to make a payment.

Recommendations

Short Term

- Remove `FixedPoint.mul` and `FixedPoint.div`, and ensure that every function uses the appropriate rounding direction
- Create two versions of `_calculateInvariant` and `_getTokenBalanceGivenInvariantAndAllOtherBalances` that round up or down, respectively, and use them appropriately.

Long term, use Echidna when designing a new pool to ensure that the swap/join/exit operations are robust against rounding issues.

9. Balancer StablePool's invariant can differ significantly from Curve's invariant

Severity: Undetermined
Difficulty: Medium
Type: Undefined Behavior
Finding ID: TOB-BALANCER-009
Target: *StableMath.sol*

Description

Balancer implements `StablePool` with a formula based on Curve's formula. The two implementations employ different orders of operations and rounding strategies, leading to different results. The impact of this difference is unclear and requires further investigation.

Curve's invariant is as follows:

```
def get_D(xp: uint256[N_COINS], amp: uint256) -> uint256:
    """
    D invariant calculation in non-overflowing integer operations
    iteratively
    A * sum(x_i) * n**n + D = A * D * n**n + D**(n+1) / (n**n * prod(x_i))
    Converging solution:
    D[j+1] = (A * n**n * sum(x_i) - D[j]**(n+1) / (n**n prod(x_i))) / (A * n**n - 1)
    """
    S: uint256 = 0

    for _x in xp:
        S += _x
    if S == 0:
        return 0

    Dprev: uint256 = 0
    D: uint256 = S
    Ann: uint256 = amp * N_COINS
    for _i in range(255):
        D_P: uint256 = D
        for _x in xp:
            D_P = D_P * D / (_x * N_COINS + 1)  # +1 is to prevent /0
        Dprev = D
        D = (Ann * S / A_PRECISION + D_P * N_COINS) * D / ((Ann - A_PRECISION) * D /
A_PRECISION + (N_COINS + 1) * D_P)
        # Equality with the precision of 1
        if D > Dprev:
            if D - Dprev <= 1:
                return D
        else:
            if Dprev - D <= 1:
                return D
    # convergence typically occurs in 4 rounds or less, this should be unreachable!
    # if it does happen the pool is borked and LPs can withdraw via `remove_liquidity`
    raise
```

Figure 9.1: Curve's invariant

Balancer's invariant is as follows:

```
function _calculateInvariant(uint256 amplificationParameter, uint256[] memory balances)
    internal
    pure
    returns (uint256)
{
    /**********************************************************************************
    // invariant                                                                      //
    // D = invariant                                                  D^(n+1)         //
    // A = amplification coefficient      A  n^n S + D = A D n^n + -----------        //
    // S = sum of balances                                             n^n P          //
    // P = product of balances                                                        //
    // n = number of tokens                                                           //
    **********************************************************************************/

    // We round up invariant.

    uint256 sum = 0;
    uint256 numTokens = balances.length;
    for (uint256 i = 0; i < numTokens; i++) {
        sum = sum.add(balances[i]);
    }
    if (sum == 0) {
        return 0;
    }
    uint256 prevInvariant = 0;
    uint256 invariant = sum;
    uint256 ampTimesTotal = Math.mul(amplificationParameter, numTokens);

    for (uint256 i = 0; i < 255; i++) {
        uint256 P_D = Math.mul(numTokens, balances[0]);
        for (uint256 j = 1; j < numTokens; j++) {
            P_D = Math.divUp(Math.mul(Math.mul(P_D, balances[j]), numTokens), invariant);
        }
        prevInvariant = invariant;
        invariant = Math.divUp(
            Math.mul(Math.mul(numTokens, invariant),
                     invariant).add(Math.mul(Math.mul(ampTimesTotal, sum), P_D)),
            Math.mul(numTokens.add(1), invariant).add(Math.mul(ampTimesTotal.sub(1), P_D))
        );

        if (invariant > prevInvariant) {
            if (invariant.sub(prevInvariant) <= 1) {
                break;
            }
        } else if (prevInvariant.sub(invariant) <= 1) {
            break;
        }
    }
    return invariant;
}
```

Figure 9.2: Balancer’s invariant, `pools/stable/StableMath.sol#L36-L84`

While Curve and Balancer use the same formula, they use different orders of operations and rounding methods. Additionally, Curve reverts if the precision loop does not converge after 255 iterations, while Balancer returns the value, potentially with a significant loss of precision.

We implemented a differential fuzzer through Echidna (described in Appendix G), which showed that the implementations can return significantly different values. For example, with 4 tokens with balances of 1; 1; 54,066; and 108,075,532 and an amplitude of 83,437,555, Curve returns 964,800, while Balancer returns 101,906,431, leading to a difference in magnitude of 10**8.

This difference merits further consideration, as it is unclear whether it could lead to unexpected arbitrage or other issues.

Exploit Scenario
Eve exploits unexpected arbitrage opportunities between Balancer and Curve, generating a profit. Bob notices the operation and loses confidence in Balancer’s arithmetic.

Recommendations
Short term, investigate the differences between the Balancer and Curve invariants, and evaluate their impacts on arbitrage opportunities and potential associated risks.

Long term, use differential fuzzing when designing a system for which another implementation exists.

13. StablePool's invariant is not monotonically increasing

Severity: Low
Type: Data Validation
Target: `StableMath.sol`

Difficulty: High
Finding ID: TOB-BALANCER-013

Description

`StableMath._calculateInvariant` returns the pool's invariant, which indicates the value of the pool. The invariant is expected to be monotonically increasing but is not.

`StableMath._calculateInvariant` returns the pool's invariant and is an approximation of the following:

$$An^n \sum x_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod x_i}$$

Figure 13.1: The pool's invariant

```
function _calculateInvariant(uint256 amplificationParameter, uint256[] memory balances)
    internal
    pure
    returns (uint256)
{
    /**********************************************************************************
    // invariant                                                                    //
    // D = invariant                                                D^(n+1)         //
    // A = amplification coefficient      A  n^n S + D = A D n^n + -----------      //
    // S = sum of balances                                            n^n P         //
    // P = product of balances                                                      //
    // n = number of tokens                                                         //
    **********************************************************************************/

    // We round up invariant.

    uint256 sum = 0;
    uint256 numTokens = balances.length;
    for (uint256 i = 0; i < numTokens; i++) {
        sum = sum.add(balances[i]);
    }
    if (sum == 0) {
        return 0;
    }
    uint256 prevInvariant = 0;
    uint256 invariant = sum;
    uint256 ampTimesTotal = Math.mul(amplificationParameter, numTokens);

    for (uint256 i = 0; i < 255; i++) {
        uint256 P_D = Math.mul(numTokens, balances[0]);
        for (uint256 j = 1; j < numTokens; j++) {
            P_D = Math.divUp(Math.mul(Math.mul(P_D, balances[j]), numTokens), invariant);
        }
        prevInvariant = invariant;
        invariant = Math.divUp(
            Math.mul(Math.mul(numTokens, invariant),
                     invariant).add(Math.mul(Math.mul(ampTimesTotal, sum), P_D)),
            Math.mul(numTokens.add(1), invariant).add(Math.mul(ampTimesTotal.sub(1), P_D))
        );

        if (invariant > prevInvariant) {
            if (invariant.sub(prevInvariant) <= 1) {
                break;
            }
        } else if (prevInvariant.sub(invariant) <= 1) {
            break;
        }
    }
    return invariant;
}
```

Figure 13.2: `pools/stable/StableMath.sol#L36-L84`

This invariant is expected to increase if the balance of the pool increases. Due to the incorrect application of `_calculateInvariant`, this assumption can be broken. As a result, the pool's invariant could decrease over time.

Exploit Scenario

Bob deploys a stable pool with 4 tokens. Their balances are 1; 2; 17,464,062,808,818,790,875; and 14,953,662,333,476,747, and the `amp` parameter is 1,001,313,087,410,729,399.

The current invariant is 6,600,767,538,511,083,210. The balance of each token increases by 1 wei. The pool invariant is now 6,600,767,538,511,083,209 (i.e., the original invariant, less 1). As a result, the pool has lost value, while the balance has increased.

Recommendations

Short term, apply the correct rounding in `StableMath._calculateInvariant.`

Long term, identify the system's invariants, and use Echidna to check their robustness.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users’ information is at risk; exploitation could pose

	reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	Commonly exploited public tools exist, or such tools can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Code Stability	Related to the recent frequency of code updates
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.

Missing	The component was missing.
Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

PoolRegistry

- **Remove the counter library.** The library is used only twice, once to read its value and once to increment it. Using a library for these purposes is cumbersome.

BasePool

- **Remove BasePoolAuthorization from the BasePool's constructor.** There is no constructor for `BasePoolAuthorization` (or `Authentication`), but users may incorrectly assume that calling it will result in initialization.
- **Consider emitting an event with the registered pool parameters in the constructor**. This will make it easy for off-chain systems to monitor the creation of new pools.

BasePoolFactory

- **Remove unused import IBasePool**. The contract does not use this interface.

StablePoolUserDataHelpers

- **Rename minBPTAmountIn as minBPTAmountOut in exactTokensInForBptOut**. The ABI-decoded argument signifies BPT out, not in.

WeightedPoolUserDataHelpers

- **Rename minBPTAmountIn as minBPTAmountOut in exactTokensInForBptOut**. The ABI-decoded argument signifies BPT out, not in.

Code Size Optimizations

The contracts are approaching the code size limit. We recommend making the following changes to reduce the contracts' code size:

- **Replace modifiers with internal calls.** The compiler generates boilerplate code for modifiers that is not optimized and leads to code duplication. Replacing modifiers with internal calls will reduce the code size in some cases (see #6584).

- **Remove `trackExempt` from AssetTransfersHandler._sendAsset.** The parameter is `false` in all of the calls (aside from those in the mock contracts) and can be removed.
- **Remove the bit mask from `PoolRegistry._getPoolAddress`.** The bitwise shifting to the right already ensures that the leading bits are zero. Moreover, the value is cast to an address that applies the same mask.

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such

cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` and `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non–token-related functions.** Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., balances[token_address][msg.sender] may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.

- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

E. Risks Associated with Arbitrary Pools

Balancer aims to accept arbitrary pools, the code of which has not been vetted by developers. Arbitrary pools introduce additional issues that could allow an attacker to steal users' funds. We recommend that users review the pool code to ensure it behaves as expected. Pools should meet the following criteria:

- **They should not be upgradeable.** Upgradeable pools have inherited risks.
- **They should be unable to self-destruct.** Destructible pools have inherited risks; for example, they may be subject to malicious upgrades through `create2`.
- **Pools should have a clear fee schema and should not allow users to steal funds using high fees.** Users must be aware of the amount of fees that pools earn on and take from transactions.
- **Pools should not have calls to `deregisterTokens`.** If a pool can call `deregisterTokens`, it may be able to trap users' funds.
- **Pool tokens' weight should be immutable and bounded.** Otherwise, an attacker could abuse updates to a token's weight to gain a profit.
- **Tokens with decimals != 18 should be handled correctly.** Tokens with different `decimals` can lead to incorrect price computations.
- **Pools should have documented parameters.** Each pool's parameters must be documented and made clear to users.
- **Pools should favor immutable parameters.** Immutable parameters reduce the risks associated with privileged users.

F. Checking `nonReentrant` Modifiers with Slither

The Balancer codebase has many functions that can interact with third-party pools. As a result, most of the functions are protected by a `nonReentrant` modifier. We used [Slither](https://github.com/crytic/slither) to identify and review the functions that are not protected by a modifier.

The following script uses a whitelist approach, in which every reachable function must be either protected with a `nonReentrant` modifier or whitelisted.

No issue was found using the script.

```python
from slither import Slither
from slither.core.declarations import Contract
from typing import List

contracts = Slither(".", ignore_compile=True)

def _check_access_controls(
    contract: Contract, modifiers_access_controls: List[str], whitelist: List[str]
):
    print(f"### Check {contract} access controls")
    no_bug_found = True
    for function in contract.functions_entry_points:
        if function.is_constructor:
            continue
        if function.view:
            continue

        if not function.modifiers or (
            not any((str(x) in modifiers_access_controls) for x in function.modifiers)
        ):
            if not function.name in whitelist:
                print(f"\t- {function.canonical_name} should have an non re-eentrant
modifier")
                no_bug_found = False
    if no_bug_found:
        print("\t- No bug found")

_check_access_controls(
    contracts.get_contract_from_name("Vault"),
    ["nonReentrant"],
    ["queryBatchSwap", "receive", "setEmergencyPeriod"],
)
```

G. Differential Fuzzing on Balancer <> Curve

The Balancer codebase implements a stable pool with a formula based on Curve's formulas. Using Echidna, we developed a differential fuzzer to fuzz test the two implementations.

The differential fuzzer runs both the Curve and Balancer invariant functions and looks for cases in which the functions return values with a difference of at least 10**8.

We discovered issue TOB-BALANCER-09 using the differential fuzzer.

```
import "./StableMath.sol";

interface Curve{

    function get_D(uint256[4] memory, uint256) external view returns(uint256);

}

contract DifferentialFuzzing is StableMath{
    using FixedPoint for uint256;

    uint256 internal constant A_PRECISION = 100; // Curve amplification precision

    Curve c;

    constructor() public{
        address curve_addr;
        bytes memory curve_bytecode =
hex"61033056600436101561000d57610326565b600035601c52600051341561002157600080fd5b63c763592c81
14156103245760006101405261018060006004818352015b60206101805102600401356101605261014080516101
605181818301101561006657600080fd5b808201905090508152505b815160010180835281141561003f575b5050
61014051151561009857600060005260206000f35b60006101605261014051610180526084356004808202821582
84830414176100bf57600080fd5b809050905090506101a0526101c0600060ff818352015b610180516101e05261
022060006004818352015b6020610220510260040135610200526101e05161018051808202821582848304141761
011557600080fd5b80905090509050610200516004808202821582848304141761013657600080fd5b8090509050
9050600181818301101561014e57600080fd5b80820190509050808061016057600080fd5b8204905090506101e0
525b81516001018083528114156100ea575b505061018051610160526101a0516101405180820282158284830414
176101a157600080fd5b809050905090506064808204905090506101e051600480820282158284830414176101cb
57600080fd5b809050905090508181830110156101e157600080fd5b808201905090506101805180820282158284
8304141761020057600080fd5b809050905090506101a05160648082101561021a57600080fd5b80820390509050
61018051808202821582848304141761023957600080fd5b8090509050905060648082049050905060056101e051
808202821582848304141761026357600080fd5b8090509050905081818301101561027957600080fd5b80820190
509050808061028b57600080fd5b82049050905061018052610160516101805111156102da576001610180516101
6051808210156102ba57600080fd5b8082039050905011156102d5576101805160005260206000f35b61030c56
5b60016101605161018051808210156102f157600080fd5b8082039050905011151561030c576101805160005260
206000f35b815160010180835281141561000d6575b505060006000fd5b505b60006000fd5b61000461033003610
0460003961000461033003600f3";


        uint size = curve_bytecode.length;

        assembly{
            curve_addr := create(0, 0xa0, size)
        }
        c = Curve(curve_addr);
    }

    uint limit = 10**8;

    function test(uint[4] memory balances, uint amp) public {
        uint curve_value = c.get_D(balances, amp * A_PRECISION);

        uint[] memory balances_dynamic_array = new uint[](4);
        balances_dynamic_array[0] = balances[1];
        balances_dynamic_array[1] = balances[1];
        balances_dynamic_array[2] = balances[2];
        balances_dynamic_array[3] = balances[3];
        uint balance_value = _calculateInvariant(amp, balances_dynamic_array);

        if(curve_value>balance_value){
            assert(curve_value - balance_value< limit);
        }
        if(curve_value<balance_value){
            assert(balance_value - curve_value< limit);
        }
    }
}
```

Figure G: Echidna-based differential fuzzer.

H. Fixed-Point Rounding Recommendations

The Balancer codebase uses fixed-point arithmetic. Our initial recommendations regarding the rounding strategies to apply to the `WeightedPool` are included below. We recommended applying a strategy of rounding down or up such that the operations are always beneficial to the pool.

The process of determining the rounding direction for each operation is described below.

Rounding Primitives

The `mul` and `div` operations are arithmetic primitives of the fixed point:

$$mulFixedPoint(a, b) \;=\; \frac{a * b + \frac{10^{**}18}{2}}{10^{**}18}$$

```
 function mul(uint256 a, uint256 b) internal pure returns (uint256) {
      uint256 c0 = a * b;
      require(a == 0 || c0 / a == b, "ERR_MUL_OVERFLOW");
      uint256 c1 = c0 + (ONE / 2);
      require(c1 >= c0, "ERR_MUL_OVERFLOW");
      uint256 c2 = c1 / ONE;
      return c2;
  }
```

Figure F.1: `math/FixedPoint.sol#L82-L89`

$$divFixedPoint(a, b) \;=\; \frac{a * 10^{**}18 + \frac{b}{2}}{b}$$

```
 function div(uint256 a, uint256 b) internal pure returns (uint256) {

      require(b != 0, "ERR_DIV_ZERO");

      uint256 c0 = a * ONE;

      require(a == 0 || c0 / a == ONE, "ERR_DIV_INTERNAL"); // mul overflow

      uint256 c1 = c0 + (b / 2);

      require(c1 >= c0, "ERR_DIV_INTERNAL"); //  add require

      uint256 c2 = c1 / b;

      return c2;

  }
```

Figure F.2: `math/FixedPoint.sol#L100-L108`

Every arithmetic operation (for both swaps and liquidity) is based on these primitives.

Fixed-Point Primitives

In non–fixed-point arithmetic, multiplication does not lose precision (if we ignore overflow). Only division operations need the two primitives to approximate results.

Rounding down is the default behavior:

$$div_{down}(a, b) = \frac{a}{b}$$

The below approximation can be used to round up a division operation. (See Number Conversion, Roland Backhouse.)

$$div_{up}(a, b) = \frac{a + b - 1}{b}$$

div_{down} and div_{up} can then be used as primitives to build the following:

- $mulFixedPoint_{up}$
- $mulFixedPoint_{down}$
- $divFixedPoint_{down}$
- $divFixedPoint_{up}$

Determining the Rounding Direction

To determine the rounding direction (i.e., up or down), one can reason out every operation's outcome.

For example, `_inGivenOut` calculates the number of `aI` In tokens that must be paid to receive `aO` Out tokens:

$$aI = bI * \left(\left(\frac{bo}{bo - a0}\right)^{\frac{w0}{wi}} - 1 \right)$$

To benefit the pool, `aI` must tend toward a high value (↗), resulting in the following:

- $bI * \left(\left(\frac{bo}{bo - a0}\right)^{\frac{w0}{wi}} - 1\right)$ must ↗
- $\left(\left(\frac{bo}{bo - a0}\right)^{\frac{w0}{wi}} - 1\right)$ must ↗
- $\left(\frac{bo}{bo - a0}\right)^{\frac{w0}{wi}}$ must ↗

  - $\frac{bo}{bo - a0}$>= 1
- $\frac{w0}{wi}$must ↗ (See the “Power Rounding” section below.)
- $\frac{bo}{bo - a0}$must ↗

So the following formula must apply:

$$aI \ = \ bI *_{\nearrow} ((\frac{bo}{bo - a0} \nearrow)^{\frac{w0}{wi} \nearrow}) \ - \ 1 \ )$$

The same analysis can be applied in all of the system’s formulas.

Power Rounding

Several operations require the computation of $C^{\frac{a}{b}}$. The following describes the rules for applying rounding on $\frac{a}{b}$:

- If c >= 1,
  - $c^{\frac{a}{b}}$ ↗ requires $\frac{a}{b}$↗ and
  - $c^{\frac{a}{b}}$ ↘ requires $\frac{a}{b}$↘.
- If c < 1,
  - $c^{\frac{a}{b}}$ ↗ requires $\frac{a}{b}$↘ and
  - $c^{\frac{a}{b}}$ ↘ requires $\frac{a}{b}$↗.

Note that this does not apply to rounding in the `pow` function (`LogExpMath.sol#L229`).

(1-x) vs. (x-1) Rounding

Several operations require the computation of $(1 \ - \ x)$or $(x \ - \ 1)$. The following describes the rules for applying the rounding:

- For $(1 - x)$,
    - $(1 - x)\nearrow$ requires $x\searrow$ and
    - $(1 - x)\searrow$ requires $x\nearrow$.
- For $(x - 1)$,
    - $(x - 1)\nearrow$ requires $x\nearrow$ and
    - $(x - 1)\searrow$ requires $x\searrow$.

Rounding Results

The following explains which rounding direction should be applied, without considering the fees.

joinPool

How much $token_{paid}$ amount must be paid to receive `poolAmountOut` of BPT tokens ($token_{paid}\nearrow$).

$$token_{paid} = poolBalanceToken *_{\nearrow} \frac{poolAmountOut}{poolTotal}\nearrow$$

exitPool

How much $token_{received}$ amount will be received in exchange for `poolAmountIn` of BPT sent ($token_{received}\searrow$).

$$token_{received} = poolBalanceToken *_{\searrow} \frac{poolAmountIn}{poolTotal}\searrow$$

_tokenInForExactBPTOut

How many tokens should be paid for `bptAmountOut` of BPT tokens ($token_{paid}\nearrow$).

$$token_{paid} = tokenBalancer *_{\nearrow}\left(\left(\frac{bptTotalSupply + bptAmountOut}{bptTotalSupply}\nearrow\right)^{\frac{1}{tokenNormalizedWeight}\nearrow} - 1\right)$$

_bptInForExactTokensOut

How much bpt_{in} that must be paid to receive $amountsOut_{t}$ of tokens ($bpt_{in}\nearrow$).

$$bpt_{in} = bptTotalSupply *_{\nearrow} (1 - \prod_{t\searrow} (1 - (\frac{amountsOut_{t}}{balance_{t}}\nearrow)^{weight_{t}})$$

_inGivenOut

How much `aI` In tokens must be paid to receive `a0` of Out tokens `( aI↗).`

$$aI = bI *_{\nearrow} ((\frac{bo}{bo - a0}\nearrow)^{\frac{w0}{wi}\nearrow}) - 1)$$

_outGivenIn

How much `a0` Out tokens that will be received when `aI` In tokens are paid (a0 ↘).

$$a0 = b0 *_{\searrow} (1 - (\frac{bi}{bi + ai}\nearrow)^{\frac{wi}{wo}\searrow})$$