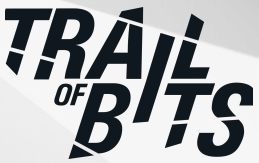

Balancer Composable Stable Pool

Security Assessment

September 22, 2022

Prepared for:
Nicolás Venturo
Balancer Labs

Prepared by: **Josselin Feist, Gustavo Grieco, and Usmann Khan**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Balancer engaged Trail of Bits to review the security of its Composable Stable Pool. From August 22 to September 2, 2022, a team of three consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed static and dynamic automated and manual testing of the target system and its codebase.

Summary of Findings

The audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of the findings and details on the notable finding are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	0
Low	1
Informational	3
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	3
Timing	1
Undefined Behavior	1

Notable Findings

A significant flaw that impacts system confidentiality, integrity, or availability is listed below.

- **TOB-BALANCER-STABLE-1: Incorrect inheritance chaining prevents from being callable**
  An incorrect chaining of calls to `super` in the `_isOwnerOnlyAction` function prevents access to the `BasePool` contract's privileged functions.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Josselin Feist, Consultant
josselin@trailofbits.com

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Usmann Khan, Consultant
usmann.khan@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 16, 2022	Pre-project kickoff call
August 29, 2022	Status update meeting #1
September 6, 2022	Delivery of report draft
September 6, 2022	Report readout meeting
September 22, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of Balancer's Composable Stable Pool. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the join and exit operations properly handled?
- Could swapping with BPT tokens result in unexpected behavior?
- Could the virtual supply be manipulated?
- Do the fees match their expected values?
- Is it possible to steal funds or lose tokens?

Project Targets

The engagement involved a review and testing of the following target.

pkg/pool-stable/contracts/ComposableStablePool.sol

Repository	https://github.com/balancer-labs/balancer-v2-monorepo/
Version	`3774c9c3ecd4dace138844f7407f85dcd04a4c77`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- `ComposableStablePool`. This contract implements a stable pool with preminted BPT and rate providers for each token that can be swapped, allowing for swaps of tokens other than stablecoins (e.g., Compound's cTokens). This pool provides support not only for swapping tokens, but also for swapping BPT. We manually reviewed the process for initializing and updating this pool's bookkeeping and for defining and executing each type of swap. We also checked how specific code in this contract overrides each feature of the base contracts (e.g., `BasePool` and `BaseGeneralPool`), looking for incomplete, unexpected, or incoherent behavior. We inspected the BPT cycle, from minting through swapping and burning. Finally, we checked how the pool behaves when emergency mode is used, ensuring that users can always remove their liquidity if needed.

- `StablePoolAmplification`. This contract uses timestamps to slowly update its amplification parameter over time. We manually reviewed this code to ensure that the amplification parameter is updated as expected without integer overflows or reverts that can block the system's normal operation. Additionally, we used automated tools to test a number of arithmetic invariants related to the computation of the amplification factor.

- `ComposableStablePoolRates`. This contract provides a token rate cache to avoid having to query the price rate for a token every time a pool needs to work with said token. We manually reviewed how rates are initialized, adjusted, queried, and flushed. Additionally, we looked for access control issues, uses of invalid or outdated values, and unexpected reverts that can block the normal operation of a pool.

- `ComposableStablePoolProtocolFees`. This contract tracks protocol fees, measuring and storing the value of the fee invariant after every join and exit operation. We manually reviewed how fees are calculated, adjusted, and stored to ensure that these processes match those indicated in the inline documentation. We also looked for the use of invalid values that could return unexpected fee amounts or that could unexpectedly revert, blocking the normal operation of a pool.

- `ComposableStablePoolStorage`. This contract performs the initial validation and storage of important state variables, such as tokens, balances, and scaling factors. It also implements specific code to calculate the virtual total supply, which should be used instead of the total supply across the codebase. We reviewed how these variables are stored and validated to ensure that they are properly updated and can never reach invalid values. We also reviewed the use of the virtual total supply

instead of the total supply in other parts of the codebase to ensure that it is used correctly.

- `ComposableStablePoolFactory`. This contract allows any user to deploy composable stable pools, using certain trusted parameters such as a particular vault address. We manually reviewed this contract to ensure that users cannot deploy malicious pools that allow them to use unexpected values in their internal parameters.

Additionally, our code coverage relied on the following assumptions:

- The arithmetic operations and invariants implemented in the `StableMath` library are correct.
- The token rate functions are monotonically increasing.
- All the non-pool components are trusted, including the Vault.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- We did not look for opportunities to conduct high-level economic attacks to subvert expected invariants of the pool.
- We did not cover the economics of the fees (i.e., we assumed that the formulas described in the documentation are correct).
- We did not look for issues that could arise from the use of the other components (e.g., the Vault) or third-party integrations (e.g., the rate provider).
- We did not look for complex arbitraging opportunities, particularly those requiring front-running of swap operations or drastic parameter changes.
- We did not cover the `StableMath` and `WordCodec` libraries, which were explicitly marked as out of scope.
- We only partially covered the base components (e.g., `BasePool`). We focused on the base components only when they were required to understand the behavior of the stable pool.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables. We used Slither to detect common issues and unreachable functions.	Appendix E
Echidna	Echidna is a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation.	Appendix D

Areas of Focus

Our automated testing and verification work focused on the internal arithmetic properties of the `StablePoolAmplification` contract.

Test Results

The results of this focused testing are detailed below.

Property	Tool	Result
The amplification parameter is bounded by `MAX_AMP * AMP_PRECISION`.	Echidna	**Passed**
There are no integer overflows in the computation of the amplification parameter.	Echidna	**Passed**

There are no divisions by zero in the computation of the amplification parameter.	Echidna	**Passed**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase contains numerous numerical computations, most of which are performed with safe arithmetic or contain explicit overflow detection code. However, in some cases, invalid or unexpected values can still block certain pool operations (TOB-BALANCER-STABLE-4).	**Satisfactory**
Auditing	The contracts include a sufficient number of events to monitor their behavior off-chain. However, the Balancer team did not provide a monitoring plan, and it is unclear whether such a system or an incident response plan exists.	**Moderate**
Authentication / Access Controls	The system suffers from access controls that are separated across multiple contracts through the use of chained functions. This pattern makes it difficult to easily distinguish the owner-accessible functions from those that are not (TOB-BALANCER-STABLE-1).	**Moderate**
Complexity Management	The functions and contracts are organized and scoped appropriately and contain inline documentation that explains their workings. However, the inheritance structure is larger and more complex than necessary, resulting in code that is difficult to maintain and audit (TOB-BALANCER-STABLE-3). Additionally, data validation and transformation is done in several layers, making it difficult to follow. We detail our recommendations on complexity management in appendix F.	**Weak**

Decentralization	The number of actions that privileged actors can take is limited and easy to identify. However, the system would benefit from better documentation on these actors. The system relies on dual authorization—the pool's owner and the authorizer. The authorizer was not in scope for this review, so further investigation to evaluate this aspect of the system is required.	**Further Investigation Required**
Documentation	The codebase contains very detailed inline documentation, which sometimes serves as specification. However, important invariants are not centralized and are often difficult to find, resulting in a codebase that is difficult to review; additionally, auditing such a codebase tends to result in the discovery of false-positive issues, hindering the auditing process.	**Moderate**
Low-Level Manipulation	No assembly or low-level manipulation is present in the in-scope parts of the codebase.	**Not Considered**
Testing and Verification	The codebase contains a number of unit and integration tests. However, the tests are insufficient for catching high-severity issues such as TOB-BALANCER-STABLE-1, which indicates that the test suite should be improved. Moreover, the codebase would benefit from advanced testing techniques such as fuzzing.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Incorrect inheritance chaining prevents privileged functions from being callable	Access Controls	**High**
2	ComposableStablePoolStorage._MIN_TOKENS shadows BasePool._MIN_TOKENS	Data Validation	**Informational**
3	Excessive use of complex coding patterns	Undefined Behavior	**Informational**
4	Risk of division-by-zero error in protocol fee calculation	Data Validation	**Low**
5	Token rate update allows for arbitrage opportunity	Timing	**Undetermined**
6	Packing state variables in ProtocolFeeCache could make read operations more costly	Data Validation	**Informational**

Detailed Findings

1. Incorrect inheritance chaining prevents privileged functions from being callable

Severity: **High**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BALANCER-STABLE-1
Target: `ComposableStablePool.sol`, `StablePoolAmplification.sol`, `BasePool.sol`	

Description

An incorrect chaining of the `super` calls in the `_isOwnerOnlyAction` function prevents access to the `BasePool` contract's privileged functions.

The pool follows a complex access control pattern that requires derived contracts to override the `BasePool` contract's `_isOwnerOnlyAction` function in order to specify their owner-only functions:

```
function _isOwnerOnlyAction(bytes32 actionId)
    internal
    view
    virtual
    override(
        // Our inheritance pattern creates a small diamond that requires explicitly
listing the parents here.
        // Each parent calls the `super` version, so linearization ensures all
implementations are called.
        BasePool,
        ComposableStablePoolProtocolFees,
        StablePoolAmplification,
        ComposableStablePoolRates
    )
    returns (bool)
{
    return super._isOwnerOnlyAction(actionId);
}
```

Figure 1.1: ComposableStablePool.sol#L974-L980

Every `_isOwnerOnlyAction` call must also call `super` to explore the access controls of all the contracts in the inheritance tree. However, the `StablePoolAmplification` and `BasePool` contracts do not call `super`, as they do not inherit from contracts that call `_isOwnerOnlyAction`:

```
function _isOwnerOnlyAction(bytes32 actionId) internal view virtual override returns
(bool) {
    return
        (actionId == getActionId(this.startAmplificationParameterUpdate.selector))
||
        (actionId == getActionId(this.stopAmplificationParameterUpdate.selector));
}
```

Figure 1.2: StablePoolAmplification.sol#L193-L197

```
function _isOwnerOnlyAction(bytes32 actionId) internal view virtual override returns
(bool) {
    return
        (actionId == getActionId(this.setSwapFeePercentage.selector)) ||
        (actionId == getActionId(this.setAssetManagerPoolConfig.selector));
}
```

Figure 1.3: BasePool.sol#L264-L268

This means that the `ComposableStablePool` contract's inheritance tree has a diamond problem: chaining `super` from `ComposableStablePool` will not explore the privileged functions in `BasePool`. As a result, the system's owner cannot call any of the `BasePool` contract's privileged functions.

Appendix C provides a simplification of the codebase to highlight the issue.

Exploit Scenario

Bob is the pool's owner. He wants to change the swap fee percentage by calling `setSwapFeePercentage`, but he cannot.

Recommendations

Short term, implement an `_isOwnerOnlyAction` function that always returns `false` in the `BasePoolAuthorization` contract, and have `_isOwnerOnlyAction` in `StablePoolAmplification` and `BasePool` call `super._isOwnerOnlyAction()`.

Long term, document and test the invariants that must hold across the contract inheritance tree. Consider moving the access controls of each class into the class itself.

2. ComposableStablePoolStorage._MIN_TOKENS shadows BasePool._MIN_TOKENS

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BALANCER-STABLE-2
Target: `BasePool.sol`, `ComposableStablePoolStorage.sol`	

Description

The `ComposableStablePoolStorage._MIN_TOKENS` variable shadows the `BasePool._MIN_TOKENS` variable. This pattern is error-prone and could result in the misuse of the variables.

We classify this issue as one of informational severity, as both variables are currently private and have the same value.

```
uint256 private constant _MIN_TOKENS = 2;
```

Figure 2.1: `BasePool.sol#L59`

```
uint256 private constant _MIN_TOKENS = 2;
```

Figure 2.2: `ComposableStablePoolStorage.sol#L37`

Exploit Scenario

Bob updates `ComposableStablePoolStorage._MIN_TOKENS` to 5. This change is not reflected in `BasePool._MIN_TOKENS`, causing the checks on the minimum number of tokens in `BasePool` to fail.

Recommendations

Short term, remove `ComposableStablePoolStorage._MIN_TOKENS`.

Long term, add Slither to the CI. Slither caught this issue.

3. Excessive use of complex coding patterns

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-BALANCER-STABLE-3
Target: All the pool-related files	

Description

The codebase uses the following complex patterns excessively, making the code overall more difficult to follow and to review:

- Excessive inheritance (the use of abstract contracts and method overrides)
- Excessive use of internal calls
- Data validation that is split across multiple components

For example, requests are dispatched through the `joinSwap` and `exitSwap` operations using the following pattern:

- A ternary operator
- An internal call to a function that contains only another ternary operator
- The execution of the targeted code

```
(uint256 amountCalculated, uint256 postJoinExitSupply) = registeredIndexOut ==
getBptIndex()
    ? _doJoinSwap(
        isGivenIn,
        swapRequest.amount,
        balances,
        _skipBptIndex(registeredIndexIn),
        currentAmp,
        preJoinExitSupply,
        preJoinExitInvariant
    )
    : _doExitSwap(
        isGivenIn,
        swapRequest.amount,
        balances,
        _skipBptIndex(registeredIndexOut),
        currentAmp,
        preJoinExitSupply,
```

```
        preJoinExitInvariant
    );
```

Figure 3.1: ComposableStablePool.sol#L324-L342

```
function _doJoinSwap(
    bool isGivenIn,
    uint256 amount,
    uint256[] memory balances,
    uint256 indexIn,
    uint256 currentAmp,
    uint256 virtualSupply,
    uint256 preJoinExitInvariant
) internal view returns (uint256, uint256) {
    return
        isGivenIn
            ? _joinSwapExactTokenInForBptOut(
                amount,
                balances,
                indexIn,
                currentAmp,
                virtualSupply,
                preJoinExitInvariant
            )
            : _joinSwapExactBptOutForTokenIn(
                amount,
                balances,
                indexIn,
                currentAmp,
                virtualSupply,
                preJoinExitInvariant
            );
}
```

Figure 3.2: ComposableStablePool.sol#L362-L389

The internal functions could be removed and replaced with direct `if/else` statements:

```
bool isRegisteredIndex = registeredIndexOut == getBptIndex()
if( isRegisteredIndex && isGivenIn) {

}
else if(isRegisteredIndex &&  !isGivenIn) {

}
...
```

Figure 3.3: Example of if/else statements pattern

The `ensureInputLengthMatch` function is another example of the codebase's heavy use of internal calls that impede code reviews:

```
function ensureInputLengthMatch(uint256 a, uint256 b) internal pure {
    _require(a == b, Errors.INPUT_LENGTH_MISMATCH);
}
```

Figure 3.4: helpers/InputHelpers.sol#L21-L23

The `ensureInputLengthMatch` function calls an internal function:

```
function _revert(uint256 errorCode) pure {
    _revert(errorCode, 0x42414c); // This is the raw byte representation of "BAL"
}
```

Figure 3.5: helpers/InputHelpers.sol#L21-L23

The internal function called by `ensureInputLengthMatch` then calls another large internal function, which includes assembly code. As a result, any call to `ensureInputLengthMatch` leads to three internal calls, which could have been an inlined equality check instead.

One of the reasons for using these patterns is to reduce code duplication; however, due to its high complexity, the codebase still contains code duplication despite the use of these patterns. Consider the following example of duplicate code between the `ComposableStablePoolStorage` and `ComposableStablePoolRates` contracts:

```
for (
    bptIndex = params.registeredTokens.length - 1;
    bptIndex > 0 && params.registeredTokens[bptIndex] > IERC20(this);
    bptIndex--
) {
    // solhint-disable-previous-line no-empty-blocks
}
_bptIndex = bptIndex
```

Figure 3.6: ComposableStablePoolStorage.sol#L126-L133

```
IERC20[] memory registeredTokens = _insertSorted(rateParams.tokens, IERC20(this));
uint256 bptIndex;
for (
    bptIndex = registeredTokens.length - 1;
    bptIndex > 0 && registeredTokens[bptIndex] > IERC20(this);
    bptIndex--
) {
    // solhint-disable-previous-line no-empty-blocks
}
```

Figure 3.7: ComposableStablePoolRates.sol#L59-L67

Reviewing this codebase was much more difficult than it would have been with a flatter contract inheritance pattern, fewer internal calls, and more centralized data validation.

Recommendations

Short term, create diagrams showing the codebase's call flows, and try to apply data validation at similar levels.

Long term, consider refactoring the pool, the inheritance tree, and the functions' dependencies overall to reduce the codebase's complexity (see Appendix F).

4. Risk of division-by-zero error in protocol fee calculation

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BALANCER-STABLE-4
Target: `ComposableStablePoolProtocolFees.sol`, `ProtocolFeeCache.sol`	

Description

Because of missing validation in the protocol fee calculation, an invalid value could result in a division-by-zero error, blocking users from performing join or exit operations.

On every join and exit operation, the pool calculates the protocol fees. This calculation is performed using a base percentage, as described in the following inline documentation:

```
    function _calculateAdjustedProtocolFeeAmount(uint256 supply, uint256
basePercentage)
        internal
        pure
        returns (uint256)
    {
        // Since this fee amount will be minted as BPT, which increases the total
supply, we need to mint
        // slightly more so that it reflects this percentage of the total supply
after minting.
        //
        // The percentage of the Pool the protocol will own after minting is given
by:
        // `protocol percentage = to mint / (current supply + to mint)`.
        // Solving for `to mint`, we arrive at:
        // `to mint = current supply * protocol percentage / (1 - protocol
percentage)`.
        //
        return supply.mulDown(basePercentage).divDown(basePercentage.complement());
    }
```

Figure 4.1: The `_calculateAdjustedProtocolFeeAmount` function in `pool-stable/contracts/ComposableStablePoolProtocolFees.sol#L319-L336`

The base percentage values are obtained directly from the protocol fee provider contracts, in which any `uint64` value can be used:

```
    function _updateProtocolFeeCache(IProtocolFeePercentagesProvider
protocolFeeProvider, uint256 feeType) private {
        uint256 currentValue = protocolFeeProvider.getFeeTypePercentage(feeType);
```

```
        if (feeType == ProtocolFeeType.SWAP) {
            _cache.swapFee = currentValue.toUint64();
        } else if (feeType == ProtocolFeeType.YIELD) {
            _cache.yieldFee = currentValue.toUint64();
        } else if (feeType == ProtocolFeeType.AUM) {
            _cache.aumFee = currentValue.toUint64();
        } else {
            _revert(Errors.UNHANDLED_FEE_TYPE);
        }

        emit ProtocolFeePercentageCacheUpdated(feeType, currentValue);
    }
```

Figure 4.2: The `_updateProtocolFeeCache` function in `pool-utils/contracts/ProtocolFeeCache.sol#L136-L150`

However, if the percentage is invalid (e.g., a value over 100%), the `complement` operation returns `0`:

```
    /**
     * @dev Returns the complement of a value (1 - x), capped to 0 if x is larger
than 1.
     *
     * Useful when computing the complement for values with some level of relative
error, as it strips this error and
     * prevents intermediate negative values.
     */
    function complement(uint256 x) internal pure returns (uint256) {
        return (x < ONE) ? (ONE - x) : 0;
    }
```

Figure 4.3: The `complement` function in `solidity-utils/contracts/math/FixedPoint.sol#L152-L160`

Therefore, if the base percentage value is larger than expected, a division-by-zero error will occur, blocking the expected user operations.

Exploit Scenario

Alice deploys a pool. The contract used to provide protocol fees initially works as expected; however, at some point, it triggers a bug that causes it to return invalid fee values. The resulting division-by-zero error causes the fee calculation to revert, preventing users from performing join or exit operations.

Recommendations

Short term, add a check to disallow the use of invalid or extreme values for the base percentage values used to calculate protocol fees.

Long term, use Echidna to uncover corner cases or unexpected reverts in the codebase.

5. Token rate update allows for arbitrage opportunity

Severity: **Undetermined**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-BALANCER-STABLE-5
Target: `ComposableStablePoolRates.sol`	

Description

Anyone can update the rate of a given token. An update to a token's rate can create arbitrage opportunities that may allow a user to make a profit from the pool.

Some tokens could increase in value due to their compounding nature (e.g., cDAI). For such tokens, a trusted provider can be queried to obtain the current value of the token. Anyone can trigger an update of that value by calling `updateTokenRateCache`:

```
function updateTokenRateCache(IERC20 token) external {
    uint256 index = _getTokenIndex(token);

    IRateProvider provider = _getRateProvider(index);
    _require(address(provider) != address(0),
Errors.TOKEN_DOES_NOT_HAVE_RATE_PROVIDER);
    uint256 duration = _tokenRateCaches[index].getDuration();
    _updateTokenRateCache(index, provider, duration);
}
```

Figure 5.1: ComposableStablePoolRates.sol#L158-L165

A change in the price of such a token could create unexpected arbitrage opportunities, allowing an attacker to drain part of the pool's funds.

We classified this issue as one of undetermined severity for the following reasons:

- The rates are supposed to increase slowly, limiting the impact of the issue.
- The impact of the issue depends on the system's on-chain parameters, which were out of scope for this audit and might change over time.

A similar issue is present in the amplification parameter: a large shift in the amplification parameter could result in unexpected arbitrage opportunities. However, the risks are reduced by the use of a linear increase.

Exploit Scenario

- Compound has an issue that prevents the token rate of its token from being updated in the Balancer pool for one week.
- The Balancer pool has a Compound token whose rate will increase by 1%.
- Eve notices that the pool has not yet updated this Compound token's rate.
- Eve uses a flash loan to make three transactions:
    - One trade for $10,000,000 worth of the Compound token
    - One transaction to update the rate of the Compound token in the Balancer pool
    - One final transaction to revert the original trade
- As a result, Eve made a profit from the arbitrage opportunity created by the update.

Recommendations

Short term, implement a mechanism to monitor token rate increases, and create an incident response plan to define a process for reacting to unexpected increases in token rates.

Long term, identify all the risks related to the integration of third-party components, and ensure that a process is in place to detect and react to compromised or malfunctioning components.

References

- A 2022 Compound Labs tweet announcing a price feed issue
- Curve Vulnerability Report

6. Packing state variables in ProtocolFeeCache could make read operations more costly

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALANCER-STABLE-6
Target: `ComposableStablePool.sol`	

Description

The `ProtocolFeeCache` contract's state variables are packed to save storage slots. This pattern makes the code more difficult to review and may be more expensive to run, depending on the use of the derived contracts.

`ProtocolFeeCache` stores three fees:

```
struct FeeTypeCache {
    uint64 swapFee;
    uint64 yieldFee;
    uint64 aumFee;
}

FeeTypeCache private _cache;
```

Figure 6.1: `ProtocolFeeCache.sol#L46-L52`

The fee values are converted from `uint64` to `uint256` and are updated individually:

```
function _updateProtocolFeeCache(IProtocolFeePercentagesProvider
protocolFeeProvider, uint256 feeType) private {
    uint256 currentValue = protocolFeeProvider.getFeeTypePercentage(feeType);

    if (feeType == ProtocolFeeType.SWAP) {
        _cache.swapFee = currentValue.toUint64();
    } else if (feeType == ProtocolFeeType.YIELD) {
        _cache.yieldFee = currentValue.toUint64();
    } else if (feeType == ProtocolFeeType.AUM) {
        _cache.aumFee = currentValue.toUint64();
    } else {
        _revert(Errors.UNHANDLED_FEE_TYPE);
    }
```

Figure 6.2: `ProtocolFeeCache.sol#L136-L147`

The values are converted to `uint256` because updating `uint64` packed state variables is generally more expensive. This is because additional operations are needed:

- An additional SLOAD operation is needed to update `uint64` packed variables, as there is no SSTORE_X opcode.
- Mask operations are needed to manipulate `uint64` variables.

While this coding pattern may appear to save gas, it actually increases the gas cost within the context of `ProtocolFeeCache`.

However, depending on the use of `ProtocolFeeCache`, this optimization might be cheaper for read operations in the derived contracts. This is because EIP-2929 makes it cheaper to access the same storage slot multiple times in the same transaction. The `ComposableStablePoolProtocolFees` contract leverages this EIP-2929 feature by accessing both the swap and yield fees in one transaction, but the `ManagedPool` contract accesses only the swap fee.

In summary, an undocumented and complex optimization is implemented in the base contract, `ProtocolFeeCache`. This optimization is more costly within the context of the base contract, but it may be cheaper in the derived contracts, depending on their use.

Recommendations
Short term, document the optimization in `ProtocolFeeCache`.

Long term, carefully review optimizations before implementing them.

Summary of Recommendations

Trail of Bits recommends that Balancer address the findings detailed in this report and take the following additional steps prior to re-deployment:

- Refactor the pool, the inheritance tree, and the functions' dependencies overall to reduce the codebase's complexity (see appendix F).
- Document and test the invariants that must hold across the contract inheritance tree. Consider moving the access controls of each class into the class itself.
- Incorporate tooling into the development process.
    - Use Slither to catch possible issues before deployment.
    - Use Echidna to uncover corner cases or unexpected reverts in the codebase.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

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

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Proof of Concept for TOB-BALANCER-STABLE-1

To demonstrate the flaw described in TOB-BALANCER-STABLE-1, Trail of Bits created a simplified version of the contract inheritance tree and the use of `_isOwnerOnlyAction`:

```
pragma experimental ABIEncoderV2;

abstract contract BasePoolAuthorization {

    string[] public logs;

    function _isOwnerOnlyAction() internal virtual returns (bool);

}

abstract contract RecoveryMode is BasePoolAuthorization {

}

abstract contract ProtocolFeeCache is RecoveryMode {

}


abstract contract BasePool is BasePoolAuthorization,RecoveryMode {
    function _isOwnerOnlyAction() internal virtual override returns (bool) {
        logs.push("BasePool");
    }
}

abstract contract ComposableStablePoolStorage is BasePool {

}

abstract contract ComposableStablePoolRates is ComposableStablePoolStorage {

    function _isOwnerOnlyAction() internal virtual override returns (bool) {
        logs.push("ComposableStablePoolRates");
        super._isOwnerOnlyAction();
    }
}

abstract contract ComposableStablePoolProtocolFees is
    ComposableStablePoolStorage,
    ComposableStablePoolRates,
    ProtocolFeeCache
{

    function _isOwnerOnlyAction() internal virtual
        override(
            BasePool,
```

```
            BasePoolAuthorization,
            ComposableStablePoolRates
        )
        returns (bool)
    {
        logs.push("ComposableStablePoolProtocolFees");
        super._isOwnerOnlyAction();
    }
}

abstract contract StablePoolAmplification is BasePoolAuthorization {


    /**
     * @dev Overrides only owner action to allow setting the cache duration for the
token rates
     */
    function _isOwnerOnlyAction() internal virtual override returns (bool) {
        logs.push("StablePoolAmplification");
    }
}
abstract contract BaseGeneralPool is BasePool {

}


contract ComposableStablePool is
    BaseGeneralPool,
    StablePoolAmplification,
    ComposableStablePoolRates,
    ComposableStablePoolProtocolFees
{

    constructor() public{
        _isOwnerOnlyAction();
    }


    function _isOwnerOnlyAction()
        internal
        virtual
        override(
            // Our inheritance pattern creates a small diamond that requires
explicitly listing the parents here.
            // Each parent calls the `super` version, so linearization ensures all
implementations are called.
            BasePool,
            ComposableStablePoolProtocolFees,
            StablePoolAmplification,
            ComposableStablePoolRates
        )
        returns (bool)
    {
```

```
        super._isOwnerOnlyAction();
        logs.push("ComposableStablePool");
    }

    function get() public view returns(string[] memory){
        return logs;
    }

}
```

Figure C.1: Simplification of the contract inheritance tree

Calling `get()` returns the following contracts:

- `ComposableStablePoolProtocolFees`
- `ComposableStablePoolRates`
- `StablePoolAmplification`
- `ComposableStablePool`

`BasePool` is missing from the inheritance tree.

D. Echidna Properties for StablePoolAmplification

During this audit, Trail of Bits used Echidna to test a number of arithmetic properties for the `StablePoolAmplification` contract. To detect assertion failures, we applied the following patch:

```
--- a/pkg/pool-stable/contracts/StablePoolAmplification.sol
+++ b/pkg/pool-stable/contracts/StablePoolAmplification.sol
@@ -70,6 +70,7 @@ abstract contract StablePoolAmplification is BasePoolAuthorization
{
         )
     {
         (value, isUpdating) = _getAmplificationParameter();
+        assert(value <= Errors.MAX_AMP * StableMath._AMP_PRECISION);
         precision = StableMath._AMP_PRECISION;
     }

@@ -91,8 +92,11 @@ abstract contract StablePoolAmplification is
BasePoolAuthorization {
             // This also means that the following computation will never revert nor
yield invalid results.
             if (endValue > startValue) {
                 value = startValue + ((endValue - startValue) * (block.timestamp -
startTime)) / (endTime - startTime);
+                assert(value >= startValue);
             } else {
+                assert(endTime != startTime);
                 value = startValue - ((startValue - endValue) * (block.timestamp -
startTime)) / (endTime - startTime);
+                assert(value <= startValue);
             }
         } else {
             isUpdating = false;
```

Figure D.1: The patch to add an assertion statement that checks arithmetic invariants in `StablePoolAmplification`

To allow Echidna to test such assertions, we implemented a mock contract (`test/EchidnaStablePoolAmplification.sol`):

```
pragma solidity ^0.7.0;
pragma experimental ABIEncoderV2;

import "../StablePoolAmplification.sol";

import "@balancer-labs/v2-interfaces/contracts/vault/IVault.sol";

contract EchidnaStablePoolAmplification is StablePoolAmplification {
    constructor()
        StablePoolAmplification(StableMath._MIN_AMP)
```

```
        BasePoolAuthorization(msg.sender)
        Authentication(bytes32(uint256(address(this))))
    {}

    function getVault() public view returns (IVault) {
        return IVault(0x0);
    }

    function _getAuthorizer() internal view override returns (IAuthorizer) {
        return IAuthorizer(0x0);
    }
}
```

Figure D.2: The `EchidnaStablePoolAmplification` contract

Our fuzzer can be executed using the following command:

```
$ echidna-test . --contract EchidnaStablePoolAmplification --test-mode assertion
```

E. Slither Script to Detect Unreachable Functions

This appendix provides a Slither script that checks for the use of the `ComposableStablePool._scalingFactor` function, which is intended to be unreachable.

According to the code comment below, `ComposableStablePool._scalingFactor` should never called:

```
function _scalingFactor(IERC20) internal view virtual override returns (uint256) {
    // We never use a single token's scaling factor by itself, we always process the
entire array at once.
    // Therefore we don't bother providing an implementation for this.
    _revert(Errors.UNIMPLEMENTED);
```

Figure E.1: ComposableStablePoolStorage.sol#L248-L252

Calls to this function will always revert. The following script uses Slither to check that this function is never called:

```
from slither import Slither
from slither.core.declarations import Function

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

for compilation_unit in sl.compilation_units:
    contracts = compilation_unit.get_contract_from_name("ComposableStablePool")

    assert len(contracts) <= 1

    if len(contracts) == 1:
        composableStablePool = contracts[0]
        print(f"Checking {composableStablePool}")
        scalingFactor =
composableStablePool.get_function_from_signature("_scalingFactor(address)")
        assert scalingFactor
        if scalingFactor.reachable_from_functions:
            print(
                f"{scalingFactor} is reachable from {[str(f) for f in
func.reachable_from_functions]}"
            )
        else:
            print(f"{scalingFactor} is not reachable")
```

Figure E.2: check_scaling_factor.py

We recommend adding the script to the project's CI to prevent future bugs that would allow this function to be called from being introduced.

Note that the script has two limitations:

- Slither does not work out-of-the-box on this codebase because of the use of the ternary operator in `ComposableStablePool.sol#L313`. Replacing the ternary operator with an `if/then/else` statement would allow the script the run.
- The script does not check for the use of function pointers.

F. Design Recommendations

Because of the current structure of the code and the organization of the contracts, the project is very complex. In addition to the inherent complexity of the Balancer system, this makes the codebase difficult to review and increases the likelihood of mistakes. This appendix contains guidelines on how to reduce this complexity; it is an extension of the recommendations provided for TOB-BALANCER-STABLE-3.

Code Structure

- **Replace the join/exit dispatcher in the `ComposableStablePoolStorage` contract with an `if/then/else` statement.** The current pattern—using ternary operators and internal functions—makes the code difficult to understand.

Data Manipulation

- **Standardize the code responsible for input validation and transformation, by either moving it closer to the relevant functions or implementing it as a general pre-processing step.** Many functions have inputs with several attributes that must be validated or transformed. Currently, these attributes are checked at different points in the input pipeline. For example, a value may be multiplied by a scaling factor and then undergo several checks (e.g., a check ensuring the value is a positive number) all before a given function is reached. As a result of this data manipulation pattern, it is difficult to discern the checks or transformations that an input has undergone before reaching a particular point in the codebase.

Documentation

- **Create diagrams and graphs to highlight the project's execution logic and underlying assumptions.** Such diagrams and graphs should describe how the contracts interact with each other (e.g., interactions between the vault and the pool) and the underlying assumptions of the various interactions (e.g., the vault checks that `tokenIn != tokenOut`).

- **Clearly document invariants for all logical components.** While the codebase has extensive inline documentation, the restrictions that should apply to certain operations may be unclear. For example, pools' join and swap operations have an important property that must be held, but this property is documented only in the Vault code. This inconsistent documentation makes it difficult to verify whether functions are implemented as intended and makes implementing new code error-prone. Considering the complex inheritance structure, we recommend linking related documentation to ease the review of all components. This would allow developers and code reviewers to better understand the code, even when they only have a partial view of the codebase.

- **Document the arithmetic formulas with examples containing concrete values.** Having examples with concrete values will help developers and code reviewers to understand the arithmetic operations and validate that they are implemented as expected. In particular, documentation on the formulas related to the fees, the exempt token, and changes to invariants would benefit from concrete examples.

Inheritance

- **Consider using libraries rather than the current inheritance structure.** The current inheritance structure is difficult to follow and is error-prone (see TOB-BALANCER-STABLE-1). The code could be refactored to remove the use of inheritance and to instead rely on libraries for common operations. If libraries cannot be used, consider using well-documented templates with a clear procedure for updates instead. Templates require copying and pasting code and will make the development process more heavy; however, they will reduce code reviewing efforts and the underlying risks.

- **Limit function flows to either a base contract or a derived contract.** Balancer pool operations are defined in base contracts. Derived contracts often override some, but not all, of these operations. This results in a pattern in which function calls bounce between the base and derived contract, making the code difficult to follow.