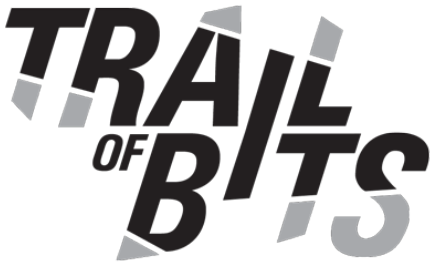

Linear and Phantom Pools

Security Assessment

December 22, 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

Natalie Chin | *Trail of Bits*
natalie.chin@trailofbits.com

Changelog:

October 28, 2021:	Initial report delivered
December 22, 2021:	Added Appendix G with additional review

Executive Summary

From September 27 to October 8, 2021, Balancer engaged Trail of Bits to review the security of its linear and phantom pools. Trail of Bits conducted this assessment over four person-weeks, with three engineers working from commits `26a1dc64` and `ceba3781` of the Balancer repository.

During the first week of the assessment, we focused on gaining an understanding of the `LinearPool` contract, its interactions with the vault, and its architecture. During the second week, we analyzed the linear pool arithmetic and reviewed the `StablePhantomPool` contract, focusing on its fee structure and integration with the stable pool mechanism. In addition to performing a manual review, we developed a Slither script to use in our review of the `authenticate` modifier's access controls (see Appendix C). We also wrote Echidna properties to fuzz the `LinearPool`'s arithmetic invariants, as detailed in the Automated Testing and Verification section.

Our review resulted in six findings, including two of high severity. The most significant finding is related to the use of an incorrect rounding direction in swap operations, which could enable an attacker to obtain free BPT when first joining a pool. The second high-severity finding involves potential arbitrage opportunities stemming from parameter updates. Other issues include the shadowing of state variables and additional risks associated with arithmetic rounding.

In addition to the security findings, we identified code quality issues not related to any particular vulnerability, which are discussed in Appendix D. Appendix E provides guidance on interactions with arbitrary ERC20 tokens.

Overall, the Balancer system adheres to smart contract best practices. Additionally, Balancer has devoted significant efforts to addressing the rounding issues in its other pools, which we identified as primary concerns in prior audits of those pools. However, because the system is highly composable, more thorough documentation on the system invariants (particularly those related to the pools' interactions with the vault) would be beneficial.

Trail of Bits recommends that Balancer take the following steps:

- Address all issues detailed in this report.
- Implement additional Echidna tests to validate the rounding of all arithmetic.
- Maintain the Slither script we developed to check functions with the `authenticate` modifier and add it to the continuous integration pipeline.
- Document the `StablePhantomPool` contract's fee mechanism and fee bookkeeping system.

During the week of December 13th 2021, Trail of Bits performed a one engineer-week review of changes made to the linear pool. Appendix G contains the details of the additional review.

Project Dashboard

Application Summary

Name	Linear and phantom pools
Versions	`26a1dc64` and `ceba3781`
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	September 27–October 8, 2021
Method	Full knowledge
Consultants Engaged	3
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■■
Total Medium-Severity Issues	0	
Total Low-Severity Issues	1	■
Total Informational-Severity Issues	2	■■
Total Undetermined-Severity Issues	1	■
Total	6	

Category Breakdown

Access Controls	2	■■
Data Validation	4	■■■■
Total	6	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory**. The project uses a robust authentication and authorization system. However, additional documentation and automated checks on the `authenticate` modifier would prevent future issues.
Arithmetic	**Moderate**. The codebase contains numerous checks of the arithmetic rounding operations; however, additional testing, including targeted fuzzing of its arithmetic, could uncover further issues. We also recommend developing additional documentation on the arbitrage risks stemming from parameter tuning.
Assembly Use	**Not Applicable.**
Code Stability	**Moderate.** Vulnerability fixes and other changes were implemented during the audit.
Decentralization	**Moderate.** The pools rely on external systems for token pricing data. Compromise of one of those external systems could enable an attacker to steal a pool's funds.
Upgradeability	**Not Applicable.**
Function Composition	**Moderate.** The codebase is well structured, and most of the functions are small and have clear purposes. However, the system's significant composability made it more difficult to review the contracts' operations.
Front-Running	**Not Considered.** The system's front-running protections are implemented by the vault, which was outside the scope of this review.
Key Management	**Not Considered.** We focused on reviewing the linear pool and phantom pool arithmetic and did not consider the deployment or storage of keys.
Monitoring	**Satisfactory.** Many functions in the contracts emit events when necessary. However, we were not provided with an incident response plan or information on the use of off-chain components in behavior monitoring.
Specification	**Moderate.** The documentation would benefit from additional detail, including on the calculation of fees and the invariants related to the pools' interactions with the vault.

Testing & Verification	**Moderate.** The system would benefit from fuzzing of the pools' arithmetic and from unit tests of all possible rounding operations.

Engagement Goals

The engagement was scoped to provide a security assessment of Balancer's linear and phantom pools.

Specifically, we sought to answer the following questions:

- Are there appropriate conditions set for all publicly callable functions?
- Are there any arithmetic overflows or underflows affecting the code?
- Could an attacker manipulate the contracts by front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Are there arithmetic rounding issues affecting the code?
- Could an attacker leverage a swap operation to steal funds?
- Could an attacker leverage the pools' interactions with the vault to engage in illicit behavior?

Coverage

`LinearPool`. We reviewed the contract for flaws that could allow an attacker to mine unearned BPT or to swap assets at no cost. As part of our review, we assessed the linear pools' arithmetic and interactions with the vault, in addition to using fuzzing to stress the `LinearMath` contract. Through a manual review and custom static analysis, we reviewed the pool initialization process and the pool owner–related access controls. We also checked the contract's use of code inherited from the `BasePool` contract and assessed the flow of the swap, join, and exit operations.

`StablePhantomPool`. Our review of this contract covered the same areas as that of the `LinearPool` contract, with the exception of the `LinearPool`'s inherited arithmetic. In addition, we reviewed the fee protocol and the contract's internal bookkeeping.

The vault and the system's other pool contracts (including the `StablePool`) were outside the scope of this assessment. We assumed the `StablePool`'s arithmetic and economics to be correct.

Automated Testing and Verification

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

- Slither, a Solidity static analysis framework
- Echidna, a smart contract fuzzer that 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, and Echidna may not randomly generate an edge case that violates a property.

Echidna Properties (LinearPool)

ID	Property	Result
1	Users cannot secure free BPT by calling `calcBptOutPerMainIn`.	PASSED
2	Users cannot secure free main tokens by calling `calcBptInPerMainOut`.	**FAILED***
3	Users cannot secure free wrapped tokens by calling `calcWrappedOutPerMainIn`.	PASSED
4	Users cannot secure free main tokens by calling `calcWrappedInPerMainOut`.	PASSED
5	Users cannot secure free BPT by calling `calcMainInPerBptOut`.	PASSED
6	Users cannot secure free main tokens by calling `calcMainOutPerBptIn`.	PASSED
7	Users cannot secure free main tokens by calling `calcMainOutPerWrappedIn`.	PASSED
8	Users cannot secure free wrapped tokens by calling `calcMainInPerWrappedOut`.	PASSED
9	Users cannot secure free BPT by calling `calcBptOutPerWrappedIn`.	PASSED
10	Users cannot secure free wrapped tokens by calling `calcBptInPerWrappedOut`.	**FAILED***

11	Users cannot secure free BPT by calling `calcWrappedInPerBptOut`.	**FAILED** (TOB-BALANCER-003)
12	Users cannot secure free wrapped tokens by calling `calcWrappedOutPerBptIn`.	PASSED
13	`toNominal` and `fromNominal` are inverse functions.	PASSED

* A call to `calcBptInPerMainOut` can result in a corner-case rounding error if the BPT supply is set to zero and the balance of the main token (`mainBalance`) is greater than zero. The same is true of calls to `calcBptInPerWrappedOut` when the BPT supply is set to zero and the balance of the wrapped token is greater than zero. As these corner cases are unlikely to occur in practice, they are not included as findings in this report.

Slither Script

ID	Property	
1	All functions that should be protected by `authenticate` are properly protected by that modifier.	APPENDIX C

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

☐ **Adjust the `LinearPool._MINIMUM_BPT` and `StablePhantomPool._MIN_TOKENS` variables such that they no longer shadow state variables.** This will help prevent the introduction of bugs in future updates and will make it easier to review the code. TOB-BALANCER-001

☐ **Improve the documentation on `_isOwnerOnlyAction`, and document—or update—the access controls on `setPaused`.** Currently, the expected access controls on the `setPaused` function are unclear. TOB-BALANCER-002

☐ **Use `divUp` in `_calcWrappedInPerBptOut` so that the function will round up and return a non-zero value if `bptOut` is less than `params.rate`.** TOB-BALANCER-003

☐ **Adjust the `_upscaleArray`, `_toNominal`, and `_fromNominal` functions to use rounding directions that benefit the pool.** This will help prevent the abuse of the functions. TOB-BALANCER-004

☐ **Document the arbitrage risks associated with parameter updates and evaluate the extent of the updates' impact on token rates.** Consider implementing a multistep process for increasing or decreasing a token rate in the event that an update causes too significant a change. TOB-BALANCER-005

☐ **Add the `whenNotPaused` modifier to all functions in the `LinearPool` and `StablePhantomPool` contracts that should not be callable when a pool is paused.** This will ensure that the functions throughout the system exhibit expected behavior when the system is paused. TOB-BALANCER-006

Long Term

☐ **Add Slither to the continuous integration pipeline.** Slither can catch certain vulnerabilities detailed in this report. TOB-BALANCER-001

☐ **Maintain the Slither script in Appendix C and use it to ensure that the access controls implemented by the `authenticate` modifier are correct.** The `authenticate` modifier's access controls are error-prone and would benefit from automated checks. TOB-BALANCER-002

☐ **Develop and maintain Echidna properties to test the pool arithmetic.** The codebase relies heavily on arithmetic primitives and would benefit from in-depth fuzzing of those primitives. TOB-BALANCER-003, TOB-BALANCER-004, TOB-BALANCER-005

☐ **Document the access controls on pool operations, along with the expected behavior of paused pools.** TOB-BALANCER-006

Findings Summary

#	Title	Type	Severity
1	Error-prone state variable shadowing	Data Validation	Low
2	authenticate modifier's error-prone and undocumented access controls	Access Controls	Informational
3	Risk of token theft caused by incorrect rounding in join operations	Data Validation	High
4	Risks associated with rounding operations	Data Validation	Undetermined
5	Risk of unexpected arbitrage stemming from parameter updates	Data Validation	High
6	Inconsistent use of whenNotPaused modifier	Access Controls	Informational

1. Error-prone state variable shadowing

Severity: Low
Type: Data Validation
Target: `LinearPool.sol`, `StablePhantomPool.sol`

Difficulty: Low
Finding ID: TOB-BALANCER-001

Description
The contracts contain two variables that shadow state variables. This pattern is error-prone and could lead to misuse of the variables.

Specifically, `LinearPool._MINIMUM_BPT` shadows `BasePool._MINIMUM_BPT`, and `StablePhantomPool._MIN_TOKENS` shadows `BasePool._MIN_TOKENS`. (Note, though, that the shadowing by `LinearPool._MINIMUM_BPT` was removed in commit `ceba3781`.)

Because the shadowed variables are declared `private`, they are not directly accessible by the contracts that inherit from `BasePool`. However, they could cause confusion during a code review or lead to the introduction of issues during future code updates.

Exploit Scenario
Bob, a Balancer developer, updates the value of `BasePool._MIN_TOKENS` from two to three. However, he incorrectly assumes that the value of `StablePhantomPool._MIN_TOKENS` will also be changed to three and does not update it. As a result, the pools may accept fewer tokens than they should.

Recommendations
Short term, adjust the `LinearPool._MINIMUM_BPT` and `StablePhantomPool._MIN_TOKENS` variables such that they no longer shadow state variables. This will help prevent the introduction of bugs in future updates and will make it easier to review the code.

Long term, add Slither to the continuous integration pipeline. Slither can catch instances of variable shadowing.

2. authenticate modifier's error-prone and undocumented access controls

Severity: Informational | Difficulty: High
Type: Access Controls | Finding ID: TOB-BALANCER-002
Target: `BasePoolAuthorization.sol` and its derived contracts

Description
The pool contracts use the `authenticate` modifier to implement access controls for privileged operations. However, functions with this modifier are checked through a complex process that is not documented and can easily be misused.

Any function with an `authenticate` modifier should be listed in the `_isOwnerOnlyAction` function; if it is not, it will be processed through the authorizer contract:

```
function _canPerform(bytes32 actionId, address account) internal view override returns
(bool) {
    if ((getOwner() != _DELEGATE_OWNER) && _isOwnerOnlyAction(actionId)) {
        // Only the owner can perform "owner only" actions, unless the owner is delegated.
        return msg.sender == getOwner();
    } else {
        // Non-owner actions are always processed via the Authorizer, as "owner only" ones
are when delegated.
        return _getAuthorizer().canPerform(actionId, account, address(this));
```

Figure 2.1: `pool-utils/contracts/BasePoolAuthorization.sol#L49-L55`

Because the contracts inherit from a base contract, they contain multiple `_isOwnerOnlyAction` functions, which makes this process error-prone. Furthermore, there is no documentation on why a function should be, or should not be, listed in `_isOwnerOnlyAction`. For example, the `setPaused` function is not listed as an "owner only" function, and it is unclear whether this omission is intended behavior.

Recommendations
Short term, improve the documentation on `_isOwnerOnlyAction`, and document—or update—the access controls on `setPaused`.

Long term, maintain the Slither script in Appendix C and use it to ensure that the access controls implemented by the `authenticate` modifier are correct.

3. Risk of token theft caused by incorrect rounding in join operations

Severity: High
Type: Data Validation
Target: `LinearMath.sol`

Difficulty: High
Finding ID: TOB-BALANCER-003

Description
The use of an incorrect rounding direction in the `_calcWrappedInPerBptOut` function, called when a user joins a pool, could facilitate token theft.

The `_calcWrappedInPerBptOut` function calculates the number of wrapped tokens that a user must deposit to receive a certain number of BPT tokens. If the current supply of BPT is zero, the function will calculate that value as follows:

$$amountIn = \frac{BPTout}{wrappedTokenRate}$$

```
function _calcWrappedInPerBptOut(
     uint256 bptOut,
     uint256 mainBalance,
     uint256 wrappedBalance,
     uint256 bptSupply,
     Params memory params
 ) internal pure returns (uint256) {
     // Amount in, so we round up overall.

     if (bptSupply == 0) {
         // Return nominal DAI
         return bptOut.divDown(params.rate);
```

Figure 3.1: `LinearMath.sol#L205-L217`

The result of this division is rounded down. As such, if `bptOut` is less than the `wrappedTokenRate` (i.e., the exchange rate between the main token and the wrapped token), `amountIn` will be zero. An attacker could exploit the rounding operation to secure free BPT when first joining a pool.

We identified this issue by using Echidna:

```
calcWrappedInPerBptOut(uint256,uint256,uint256,uint256,uint256,uint256,uint256): failed!💥
  Call sequence:

calcWrappedInPerBptOut(3000000000000000000000000,583102314353496781433650425818304575307712562108178052338714906457704206250 15,105592279847122353891090000574880347011784507359829712459296286618899579932014,93196427102733712501541925599138778760183718531697086608261722151155522322206,45736853129731350410463948916951127251277921291553696729466297459702906049361,946405413953279627645502533507547681823671634981558476861970030908528226239 06,70236602468866947119756237373294193729268226030983533197179848310569102801 11)

Event sequence: LogCall4(97459702906049361, 100000000000000000, 1408726256680316481555181, 3465656190698104137192229, "calcWrappedInPerBptOut", 0, 2233871490645770420625015, 2459296286618899579932014, 2608261722151155522322206, 0), AssertionFailed("free bptOut for
```

```
zero wrappedIn")
```

Figure 3.2: The Echidna report of this issue

Echidna, in the `LogCall4` event it emits, prints the arguments sent to the function, the name of the function, and the output:

```
event LogCall4(
    uint256 fee, uint256 rate, uint256 lowerTarget, uint256 upperTarget,
    string fnName,
    uint256 input,
    uint256 mainBalance, uint256 wrappedBalance, uint256 bptSupply,
    uint256 output
);
```

Figure 3.3: Echidna's definition of `LogCall4`

Exploit Scenario
An attacker, Eve, deploys a pool through the `PoolFactory`, the contract used to deploy and initialize a pool. The `wrappedTokenRate` is `10**18`. Eve calls the `_calcWrappedInPerBptOut` function with a `bptOut` value of `10**18-1`; as a result, when she joins the pool, she receives `10**18-1` free BPT tokens. Alice, a user, adds liquidity to the pool and receives BPT. Eve then swaps her BPT for the tokens added by Alice, stealing Alice's funds.

Recommendations
Short term, use `divUp` in `_calcWrappedInPerBptOut` so that the function will round up and return a non-zero value if `bptOut` is less than `params.rate`.

Long term, develop and maintain Echidna properties to test the pool arithmetic.

4. Risks associated with rounding operations

Severity: Undetermined
Type: Data Validation
Target: `LinearMath.sol`

Difficulty: Low
Finding ID: TOB-BALANCER-004

Description
The `LinearMath` and `LinearPool` contracts contain arithmetic functions that always round in the same direction. While we did not identify any significant risks or issues caused by their rounding operations, an attacker could use them to turn a profit.

These functions include `_upscaleArray`, which scales down all balances when performing array upscaling:

```
function _upscaleArray(uint256[] memory amounts, uint256[] memory scalingFactors) internal
view {
     for (uint256 i = 0; i < _getTotalTokens(); ++i) {
         amounts[i] = FixedPoint.mulDown(amounts[i], scalingFactors[i]);
     }
}
```

Figure 4.1: `pool-utils/contracts/BasePool.sol#L538-L542`

As a result, a linear pool may behave as though its token balance is lower than it actually is, which could enable an attacker to make profitable trades:

```
if (request.kind == IVault.SwapKind.GIVEN_IN) {
     _upscaleArray(balances, scalingFactors);
     request.amount = _upscale(request.amount, scalingFactors[indexIn]);
     uint256 amountOut = _onSwapGivenIn(request, balances, params);
     // amountOut tokens are exiting the Pool, so we round down.
     return _downscaleDown(amountOut, scalingFactors[indexOut]);
 } else {
     _upscaleArray(balances, scalingFactors);
     request.amount = _upscale(request.amount, scalingFactors[indexOut]);
     uint256 amountIn = _onSwapGivenOut(request, balances, params);
     // amountIn tokens are entering the Pool, so we round up.
     return _downscaleUp(amountIn, scalingFactors[indexIn]);
```

Figure 4.2: `LinearPool.sol#L190-L201`

The `_upscaleArray` function is used similarly in all Balancer pools.

Additionally, the `_toNominal` and `_fromNominal` functions, which perform conversions between nominal and real balances, always round up:

```
function _toNominal(uint256 amount, Params memory params) internal pure returns (uint256) {
     if (amount < (FixedPoint.ONE - params.fee).mulUp(params.lowerTarget)) {
         return amount.divUp(FixedPoint.ONE - params.fee);
     } else if (amount < (params.upperTarget - (params.fee.mulUp(params.lowerTarget)))) {
         return amount.add(params.fee.mulUp(params.lowerTarget));
```

```
        } else {
            return
                amount.add((params.lowerTarget + params.upperTarget).mulUp(params.fee)).divUp(
                    FixedPoint.ONE + params.fee
                );
        }
    }

    function _fromNominal(uint256 nominal, Params memory params) internal pure returns
(uint256) {
        if (nominal < params.lowerTarget) {
            return nominal.mulUp(FixedPoint.ONE - params.fee);
        } else if (nominal < params.upperTarget) {
            return nominal.sub(params.fee.mulUp(params.lowerTarget));
        } else {
            return
                nominal.mulUp(FixedPoint.ONE + params.fee).sub(
                    params.fee.mulUp(params.lowerTarget + params.upperTarget)
                );
```

Figure 4.3: `LinearMath.sol#L266-L289`

Exploit Scenario
Eve finds a way to abuse the rounding in a nominal balance–real balance conversion. She then leverages it to realize an illicit profit on a swap.

Recommendations
Short term, adjust the `_upscaleArray`, `_toNominal`, and `_fromNominal` functions to use rounding directions that benefit the pool. This will help prevent the abuse of the functions.

Long term, develop and maintain Echidna properties to test the linear pool arithmetic.

5. Risk of unexpected arbitrage stemming from parameter updates

Severity: High
Type: Data Validation
Target: `LinearPool.sol`, `StablePhantomPool.sol`

Difficulty: High
Finding ID: TOB-BALANCER-005

Description
Updates to certain parameters in the `LinearPool` and `StablePhantomPool` contracts can result in changes to pool invariants. A significant change could create unexpected arbitrage opportunities.

These parameters include the `wrappedTokenRate` parameter in the `LinearPool` contract and the `_rateProviders` parameter in the `StablePhantomPool` contract, which provides the price of each token tracked by the `RateProvider` contract.

Exploit Scenario
An update to the `LinearPool`'s `wrappedTokenRate` parameter causes a significant change in that invariant. Eve, an attacker who has been monitoring the blockchain for any contract changes, learns of the parameter update. She then front-runs the update and turns a profit by engaging in arbitrage.

Recommendations
Short term, document the arbitrage risks associated with parameter updates and evaluate the extent of the updates' impact on token rates. Consider implementing a multistep process for increasing or decreasing a token rate in the event that an update causes too significant a change.

Long term, develop and maintain Echidna properties to test the pool arithmetic.

References
- Curve Vulnerability Report

6. Inconsistent use of whenNotPaused modifier

Severity: Informational
Type: Access Controls
Target: `LinearPool.sol`, `StablePhantomPool.sol`

Difficulty: High
Finding ID: TOB-BALANCER-006

Description
The contracts use the `whenNotPaused` modifier to limit the swaps that can be performed when a linear or phantom pool is paused. However, this modifier is used inconsistently, and swap functions that lack the modifier will be callable when they should not be. This could prevent a pool's owner from addressing an issue in a pool.

For example, the `LinearPool`'s `_swapGivenMainIn` and `_swapGivenWrappedIn` functions have a `whenNotPaused` modifier, while other swap functions in the contract, such as `_swapGivenMainOut` and `_swapGivenWrappedOut`, do not:

```
function _swapGivenMainIn(
    SwapRequest memory request,
    uint256[] memory balances,
    Params memory params
) internal view whenNotPaused returns (uint256) {

 [...]

function _swapGivenWrappedIn(
    SwapRequest memory request,
    uint256[] memory balances,
    Params memory params
) internal view whenNotPaused returns (uint256) {

  [...]

function _swapGivenMainOut(
    SwapRequest memory request,
    uint256[] memory balances,
    Params memory params
) internal view returns (uint256) {

  [...]

function _swapGivenWrappedOut(
    SwapRequest memory request,
    uint256[] memory balances,
    Params memory params
) internal view returns (uint256) {
```

Figure 6.1: `LinearPool.sol#L246-L336`

Recommendations
Short term, add the `whenNotPaused` modifier to all functions in the `LinearPool` and `StablePhantomPool` contracts that should not be callable when a pool is paused.

Long term, document the access controls on pool operations, along with the expected behavior of paused pools.

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
Testing	Related to test methodology or test coverage
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	The flaw is commonly exploited; public tools for its exploitation exist or 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
Code Stability	Related to the recent frequency of code updates
Decentralization	Related to the existence of a single point of failure
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 control was robust, documented, automated, and comprehensive.
Satisfactory	With a few minor exceptions, the control was applied consistently.
Moderate	The control was applied inconsistently in certain areas.

Weak	The control was applied inconsistently or not at all.
Missing	The control was missing.
Not Applicable	The control is not applicable.
Not Considered	The control was not reviewed.
Further Investigation Required	The control requires further investigation.

C. Checking the Use of the authenticate Modifier

The `authenticate` modifier implements access controls through a complex process and must be manually added to the relevant functions (see TOB-BALANCER-002). Trail of Bits created a Slither script to check the use of `authenticate`.

The script searches all of the contracts for functions that have the `authenticate` modifier. If it finds one such function that is not listed in `_isOwnerOnlyAction` (i.e., its selector is not used in the function) and is not whitelisted (like `setPaused`), it prints a warning message. We recommend maintaining the script and adding it to the continuous integration pipeline.

```
from typing import List

from slither import Slither
from slither.slithir.operations import Member
from slither.core.declarations import Contract, Function, SolidityVariable
from slither.slithir.variables import Constant

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


def _check_access_controls(
    contract: Contract, owner_functions: List[Constant], whitelist: List[str]
) -> bool:
    """
    This function check if the contract has function with the authenticate that either:
    - Are not in owner_functions
    - Are not in

    :param contract:
    :param owner_functions
    :param whitelist:
    :return:
    :rtype: bool
    """
    bug_found = False

    for function in contract.functions_entry_points:
        if function.is_constructor:
            continue
        if function.view:
            continue

        if (
            "authenticate" in [m.name for m in function.modifiers]
            and Constant(function.name) not in owner_functions
            and not function.name in whitelist
        ):
            print(
                f"{function.name} has the authenticate modifier, but is not used in
_isOwnerOnlyAction"
            )
            bug_found = True
```

```
    return bug_found


def _get_owner_functions(function: Function) -> List[Constant]:
    """
    Return the list of function name that are read through this.function_name.selector

    :param function:
    :return:
    """
    ret: List[Constant] = []
    # Currently SlithIR does not have a great handling of this.function_name.selector
    # So we slip the operations into two steps:
    # - Collect all the REF_A on the form REF_B(None) -> REF_A.selector
    # - Return all the func_name on the form REF_A -> this.func_name
    ref_member_to_selector = [
        ir.variable_left
        for ir in function.slithir_operations
        if isinstance(ir, Member)
        if ir.variable_right == Constant("selector")
    ]
    for ir in function.slithir_operations:
        if (
            isinstance(ir, Member)
            and ir.lvalue in ref_member_to_selector
            and ir.variable_left == SolidityVariable("this")
        ):
            ret.append(ir.variable_right)

    return ret


def _get_function_read_in__isOwnerOnlyAction(contract: Contract) -> List[Constant]:
    """
    Return the list of function name that are read in _isOwnerOnlyAction functions

    :param contract
    :return:
    """
    ret: List[Constant] = []
    targets = [f for f in contract.functions if f.name == "_isOwnerOnlyAction"]
    for target in targets:
        ret += _get_owner_functions(target)
    return ret


def main():
    """
    Go over all the contracts and look for function that:
    - Have the authenticate modifier
    - Have not their selector used in _isOwnerOnlyAction
    - Are not whitelisted

    :return:
    :rtype:
    """
    visited: List[str] = []
    bug_found = False
    for compilation_unit in slither.compilation_units:
        for contract in compilation_unit.contracts_derived:
```

```
            if contract.name in visited:
                continue
            owner_functions = _get_function_read_in__isOwnerOnlyAction(contract)
            visited.append(contract.name)
            bug_found |= _check_access_controls(
                contract, owner_functions, ["setPaused"]
            )

    if not bug_found:
        print("No bug was found")


if __name__ == "__main__":
    main()
```

Figure C.1: The Slither script

D. Code Quality Recommendations

`LinearPool`

- **Move the comment that begins with "Fetches the current price rate from a provider…".** It should appear above the previous function in the call stack, the `LinearPool._getNewWrappedTokenRateCache()` function. This change will make it clear that `PriceRateCache` is not the contract that retrieves the wrapped token rate.

```
/**
 * @dev Fetches the current price rate from a provider and builds a new price rate cache
 */
function encode(uint256 rate, uint256 duration) internal view returns (bytes32) {
    _require(rate < 2**128, Errors.PRICE_RATE_OVERFLOW);

    // solhint-disable not-rely-on-time
    return
        WordCodec.encodeUint(uint128(rate), _PRICE_RATE_CACHE_VALUE_OFFSET) |
        WordCodec.encodeUint(uint64(duration), _PRICE_RATE_CACHE_DURATION_OFFSET) |
        WordCodec.encodeUint(uint64(block.timestamp + duration),
_PRICE_RATE_CACHE_EXPIRES_OFFSET);
}
```

Figure D.1: pkg/pool-utils/contracts/rates/PriceRateCache.sol#L56-L67

`StablePhantomPool`

- **Refactor `_onSwapGivenIn` to include `indexIn = _skipBptIndex(indexIn)` and `indexOut = _skipBptIndex(indexOut)` at the beginning of the function and remove the duplicate calls to `_skipBptIndex` in other branches.** This will reduce the likelihood of errors, as it will not be necessary to call `_skipBptIndex` each time these indexes are used.

```
function _onSwapGivenIn(
    SwapRequest memory request,
    uint256[] memory balancesIncludingBpt,
    uint256 indexIn,
    uint256 indexOut
) internal virtual override returns (uint256 amountOut) {
    _cacheTokenRatesIfNecessary();

    uint256 protocolSwapFeePercentage = _cachedProtocolSwapFeePercentage;

    // Compute virtual BPT supply and token balances (sans BPT).
    (uint256 virtualSupply, uint256[] memory balances) = _dropBptItem(balancesIncludingBpt);

    if (request.tokenIn == IERC20(this)) {
        amountOut = _onSwapTokenGivenBptIn(request.amount, _skipBptIndex(indexOut),
virtualSupply, balances);
```

Figure D.2: pkg/pool-stable-phantom/contracts/StablePhantomPool.sol#L177-L191

E. 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` or `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.

F. Documentation Improvements

The Balancer codebase is complex and highly composable, which makes the code more difficult to review and increases the likelihood of developer and user mistakes. Adding the following to the documentation would help clarify the code:

- **A diagram outlining the vault's interactions with the pools**
  - The diagram should highlight the calls between the contracts, the expected user operations, and the pools' interoperability.
- **A diagram outlining the pool contracts' inheritance and a call graph**
  - The pool contracts rely on multilayer inheritance, and certain functions have been overridden or make calls to functions from inherited contracts. The significant modularity of the pools makes internal calls particularly difficult to review.
- **A succinct comparison of the different types of pools**
  - This comparison should cover each type's purpose, interactions with other types of pools, construction, assumptions, and invariants.
- **A list of vault invariants, including the safeguards on arguments**
  - The vault performs much of the system's data validation (including the check of whether the tokens to be swapped are different). This list will help clarify which arguments are trusted and what data is checked.
- **Examples of how the planned transaction-batching functionality will work**
  - Balancer's end goal for the linear and phantom pools is for them to provide a means of batching transactions. Documentation on this planned functionality, including on the user flow and interactions, would be beneficial.

G. Additional review of the Linear Pool

During the week of December 13th 2021, Trail of Bits performed a one engineer-week review of changes made to the linear pool.

The changes introduced fixes and code improvements. Most of the fixes are related to issues on the admin-only functions, or issues related to the formulas not working correctly with respect to their expected behaviors:

- PR976: Pack targets in storage to save gas
- PR1054: Fix an incorrect rate calculation on the external function `getRate`
- PR1067: Arithmetic optimizations on fixed point operations to reduce gas
- PR1062, PR1064: Initiate the pool with a lower target of 0. This change was recommended informally during the audit by Trail of Bits. It prevents admin manipulation, and simplifies the related to-from nominal formulas
- PR1045, PR1061: Remove price rate cache, which could lead to arbitrage opportunities if the cache was not updated for a long period of time. The pool is now specialized per target (currently only with AAVE)
- PR1058: Remove unused rate parameter
- PR998, PR1029: Restrict admin-related functions to prevent fees from being stolen by the admins

The changes included better documentation. Overall, the addition of documentation and the code simplification make the code easier to follow. In addition, Trail of Bits reported:

- Additional arithmetics issues that require further investigation:
    - `fromNominal(toNominal( real) > real` can be true - this might lead the pool to think it has more balance than it actually has.
    - Users might be able to have free wrapped token out by abusing arithmetic rounding (see the example below)

```
   function testSwaps() public{
        LinearMath.Params memory p;
        p.fee = 1000000000000;
        p.lowerTarget = 3;
        p.upperTarget = 4;

        uint wrappedOut = 1;
        uint mainBalance = 1;

        uint mainIn =
LinearMath._calcMainInPerWrappedOut(wrappedOut,
mainBalance, p);
        require(mainIn == 0);
   }
```

    - Both issues seems to be related to the fees being always rounded down in `fromNominal/toNominal`.
- There is an issue remaining with PR998: the owner can still do unexpected actions with the fee, by doing a swap and changing the targets within the same transaction. The issue has a low likelihood and seems to have a low impact.
- `setTargets` could have the `whenNotPaused` modifier for consistency.
- The documentation on Notion is not fully up to date (e.g. `<=` is used in all the components of `R` and `N` when `fromNominal/toNominal` are defined).

Finally, Trail of Bits strongly recommends using Balancer's internal differential fuzzer to check the pools' arithmetics before deploying the contracts. Invariants should also be tested with the fuzzing campaign (e.g. the pool internal bookkeeping must always be below or equal to the pool's actual balance).