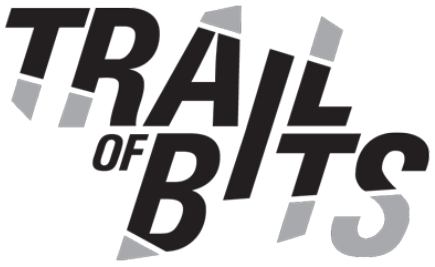

Balancer Core

Security Assessment

March 5, 2020

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

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

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

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Changelog:

February 21, 2020:	Initial report delivered
March 5, 2020:	Reviewed and updated content

Executive Summary

From January 13 through January 24, 2020, Balancer Labs engaged Trail of Bits to review the security of Balance Core. Trail of Bits conducted this assessment over the course of four person-weeks with two engineers working from 942a51e2 from the balance-core repository.

We focused on identifying important system properties to test using fuzzing (Echidna) and to verify with symbolic execution (Manticore). Specifically, we spent most of our time reviewing and testing arithmetic properties within the source code.

In total, Trail of Bits identified fifteen issues ranging from informational- to high-severity. One large category of findings allowed an attacker to obtain free tokens from pools. These issues were caused either by missing input validation during the pool creation or rounding issues during computation of the amount of tokens required to join, exit, or swap in pools. The issues of lowest severity were caused by lack of documentation for the use of tokens and their limitations when binding into a pool.

Overall, the code follows a high-quality software development standard and best practices. It has suitable architecture and is properly documented. The interactions between components are well-defined. The functions are small, with a clear purpose.

However, due to time constraints, only manual verification of essential properties related to token swapping was performed. Trail of Bits recommends performing further testing with fuzzers and symbolic executors to test extended safety and correctness properties of that important feature.

Project Dashboard

Application Summary

Name	Balancer Core
Version	942a51e202cc5bf9158bad77162bc72aa0a8afaf
Type	Smart Contracts
Platforms	Solidity

Engagement Summary

Dates	January 13 to January 24, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■ ■
Total Medium-Severity Issues	3	■ ■ ■
Total Low-Severity Issues	4	■ ■ ■ ■
Total Informational-Severity Issues	4	■ ■ ■ ■
Total Undetermined-Severity Issues	2	■ ■
Total	15	

Category Breakdown

Data Validation	7	■ ■ ■ ■ ■ ■ ■
Access Controls	1	■
Auditing and Logging	1	■
Undefined Behavior	5	■ ■ ■ ■ ■
Patching	1	■
Total	15	

Engagement Goals

The engagement was scoped to provide a security assessment of Balance Core protocol smart contracts in the `balancer-core` repository.

Specifically, we sought to answer the following questions:

- Are appropriate access controls set for the user/controller roles?
- Does arithmetic regarding pool operations hold?
- Is there any arithmetic overflow or underflow affecting the code?
- Can participants manipulate or block pool operations?
- Is it possible to manipulate the pools by front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Can participants perform denial-of-service or phishing attacks against any of the pools?

Coverage

The engagement was focused on the following components:

- **BPool and its libraries:** Pools contains the main business logic within the Balancer protocol. They allow users to bind any proper ERC20 token. We reviewed the contract's interactions with these external, untrusted token contracts, to ensure proper behavior. Once a pool is finalized or made public, we review the use of all its operations to join, exit or swap tokens.
- **BFactory:** Pools are created in a special factory contract. We reviewed the access control of this contract as well as the interaction with the pools once they are deployed.
- **BToken:** Pools mints or burns their own share tokens every time a user joins or exits the pool with tokens. This contract implements a standard ERC20 token. We verified that all the expected properties are correctly implemented.
- **Whitepaper:** the Balancer components are detailed in its whitepaper. We reviewed the document to make sure it is consistent and it does not leave any important detail unspecified.
- **Access controls.** Many parts of the system expose privileged functionality, such as setting protocol parameters or managing pools. We reviewed these functions to ensure they can only be triggered by the intended actors and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the protocol.

It is worth noting that due to time constraints, some properties related to token swapping were not covered. This includes the proper usage of `swapExactAmountIn` and `swapExactAmountOut`, as well as the interaction with the other functions.

Off-chain code components were outside the scope of this assessment.

Automated Testing and Verification

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

- Slither, a Solidity static analysis framework. Slither can statically verify algebraic relationships between Solidity variables. We used Slither to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties of the pool contract and its dependencies.
- Manticore, a symbolic execution framework. Manticore can exhaustively test security properties via symbolic execution.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode; Echidna may not randomly generate an edge case that violates a property; and Manticore may fail to complete its analysis. To mitigate these risks, we generate 50,000 test cases per property with Echidna, run Manticore for a minimum of one hour, and then manually review all results.

Automated Testing with Echidna

Echidna properties can be broadly divided in two categories: general properties of the contracts that state what users can and cannot do, and specific properties based on unit tests.

General Properties

#	Property	Result
1	An attacker cannot steal assets from a public pool.	FAILED (TOB-BL-001)
2	An attacker cannot force the pool balance to be out of sync.	PASSED
3	An attacker cannot generate free pool tokens with `joinPool`.	FAILED (TOB-BL-009)
4	Calling `joinPool-exitPool` does not lead to free pool tokens (no fee).	FAILED (TOB-BL-010)
5	Calling `joinPool-exitPool` does not lead to free pool tokens (with fee).	FAILED (TOB-BL-010)

6	Calling `exitswapExternAmountOut` does not lead to free assets.	FAILED (TOB-BL-008)

Unit-test-based Properties

#	Property	Result
7	If the controller calls `setController`, then `getController` should return the new controller.	PASSED
8	The controller cannot be changed to a null address (`0x0`).	FAILED (TOB-BL-006)
9	The controller cannot be changed by other users.	PASSED
10	The sum of normalized weight should be 1 if there are tokens binded.	FAILED (TOB-BL-012)
11	The balance of all the tokens is less than or equal to MAX_BALANCE.	FAILED (TOB-BL-014)
12	The balance of all the tokens is greater than or equal to MIN_BALANCE.	FAILED (TOB-BL-014)
13	The weight of all the tokens is less than or equal to MAX_WEIGHT.	PASSED
14	The weight of all the tokens is greater than or equal to MIN_WEIGHT.	PASSED
15	The swap fee is less than or equal to MAX_FEE.	PASSED
16	The swap fee is greater than or equal to MIN_FEE.	PASSED
17	A user can only swap in less than 50% of the current balance of `tokenIn` for a given pool.	FAILED (TOB-BL-015)
18	A user can only swap out less than 33.33% of the current balance of `tokenOut` for a given pool.	FAILED (TOB-BL-015)
19	If a token is bounded, the `getSpotPrice` should never revert.	PASSED
20	If a token is bounded, the `getSpotPriceSansFee` should never revert.	PASSED
21	Calling `swapExactAmountIn` with a small value of the same token should never revert.	PASSED

22	Calling `swapExactAmountOut` with a small value of the same token should never revert.	PASSED
23	If a user joins a pool and exits it with the same amount, the balance should remain constant.	PASSED
24	If a user joins a pool and exits it with a larger amount, `exitPool` should revert.	PASSED
25	It is not possible to bind more than MAX_BOUND_TOKENS.	PASSED
26	It is not possible to bind the same token more than once.	PASSED
27	It is not possible to unbind the same token more than once.	PASSED
28	It is always possible to unbind a token.	PASSED
29	All tokens are rebindable with valid parameters.	PASSED
30	It is not possible to rebind an unbinded token.	PASSED
31	Only the controller can bind.	PASSED
32	If a user who is not the controller tries to bind, rebind, or unbind, the operation will revert.	PASSED
33	Transferring tokens to the null address (`0x0`) causes a revert.	FAILED (TOB-BL-005)
34	The null address (`0x0`) owns no tokens.	FAILED
35	Transferring a valid amount of tokens to a non-null address reduces the current balance.	PASSED
36	Transferring an invalid amount of tokens to a non-null address reverts or returns false.	PASSED
37	Self-transferring a valid amount of tokens keeps the current balance constant.	PASSED
38	Approving overwrites the previous allowance value.	PASSED
39	The `totalSupply` is a constant.	PASSED
40	The balances are consistent with the `totalSupply`.	PASSED

Verification with Manticore

We used Manticore to verify that rounding errors cannot be used to obtain free assets while executing join and exit operations in a pool.

#	Property	Result
41	An attacker cannot generate free pool tokens with `joinPool`.	FAILED (TOB-BL-009)
42	Calling `joinPool-exitPool` does not lead to free pool tokens (no fee).	FAILED (TOB-BL-010)
43	Calling `joinPool-exitPool` does not lead to free pool tokens (with fee).	FAILED (TOB-BL-010)

Manual Verification

Trail of Bits manually reviewed the code to verify the the following properties:

#	Property	Result
44	The spot price after trading when calling `swapExactAmountIn` cannot decrease.	PASSED
45	The spot price after trading when calling `swapExactAmountOut` cannot decrease.	PASSED

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

+ **Prevent the single asset deposit/withdraw functions (`joinswapExternAmountIn`, `joinswapExternAmountOut`, `exitswapPoolAmountIn`, and `exitswapPoolAmountOut`) from being called if the pool is not finalized.** These functions allow anyone to steal the assets if the pool is not finalized **(**TOB-BL-001).

+ **Add an event to `BFactory.setBLabs`.** The lack of events makes it more difficult to track potential errors or a compromise (TOB-BL-002).

+ **Follow the same order of parameters for similar functions, such as the join and exit swap functions.** API inconsistency is error-prone and might confuse users (TOB-BL-003).

+ **Document the limitations of using pools in case token migrations**. This will help users to understand what to expect in these situations. (TOB-BL-004)

+ **Add a `require` condition in `transfer` and `transferFrom` that explicitly forbids burning tokens by transferring them to `0x0`.** This will prevent accidental burning of tokens (TOB-BL-005).

+ **Split `setController` into `setController` and `acceptController`, which will reject any null address. Add a `renounceController` function that allows you to set the controller to `0x0` if needed.** This will prevent accidental loss of controller functionality (TOB-BL-006).

+ **Prevent finalization if there are fewer than two tokens.** If a finalized pool has fewer than two tokens, anyone can receive share tokens for free (TOB-BL-007).

+ **Revert in `exitswapExternAmountOut` if `poolAmountIn` is 0**. Otherwise, the arithmetic rounding can lead to free assets (TOB-BL-008).

+ **Revert in `joinPool` if `tokenAmountIn` is zero and `tokenAmountOut` in `exitPool`.** Otherwise, the arithmetic rounding can lead to free pool tokens (TOB-BL-009).

+ **Consider preventing token with a `decimal != 18` from being bound.** Allowing tokens with different decimals complicates the computation (TOB-BL-009).

+ **Revert in `joinswapPoolAmountOut` if `calcSingleInGivenPoolOut` is zero.** Otherwise, an attacker with large funds can steal the pool's assets (TOB-BL-011).

+ **Document that the normalized sum of the weight is not always equal to 1.** Due to arithmetic rounding, this property is not enforced (TOB-BL-012).

+ **Document corner cases in the swap functions.** Due to the way integer overflows are handled, a pool with a large amount of tokens can revert when using the swap functions. (TOB-BL-013).

+ **Document how the maximum and minimum values for token balances are handled**. This will make sure users understand that these are not enforced on-chain (TOB-BL-014).

+ **Simplify the implementation of the ratio checks in the swap-in and swap-out operations using `bdiv`**. This will reduce the rounding errors in the ratio checks as well as the amount of gas used. (TOB-BL-015).

Long Term

┼ Create a state machine representation of the pool, describing each state and each function that should be callable by the different actors. Such documentation will help reviewers confirm the correct behavior of the codebase (TOB-BL-001).

┼ Monitor the contract's events to detect potential compromise. Events must be used and monitored to detect compromise (TOB-BL-002).

┼ Use Slither's printer capacity to review the contract's API. Slither can help detect API inconsistency (TOB-BL-003).

┼ Study which solutions can be implemented in case of token migration, and properly document what pool controllers and users should do. This will help users to understand what to expect in these situations. (TOB-BL-004)

┼ Use Echidna and Manticore to check that:

- **The `BToken` does not allow users to easily burn tokens by transferring them to `0x0`**. This will prevent accidental burning of tokens (TOB-BL-005).
- **The administrative addresses cannot be set to incorrect values.** This will prevent accidental loss of controller functionality (TOB-BL-006)
- **`joinPool` and `exitPool` work as expected and they cannot be used to get free assets.** This will prevent attackers from taking advantage of rounding issues (TOB-BL-007).
- **The rounding effects are below a reasonable threshold.** Multiple issues were related to arithmetic rounding; those effects must be carefully reviewed (TOB-BL-008, TOB-BL-009, TOB-BL-010, TOB-BL-011).
- **The normalized weights are correctly computed.** This will prevent incorrect computation of the required values when using the join, exit, and swap functions (TOB-BL-012).
- **The swap function does not revert when handling a large amount of tokens.** It will unexpectedly revert when calling swap functions (TOB-BL-013).
- **Pool limits are correctly enforced**. This will prevent pools with invalid or unexpected internal states (TOB-BL-014, TOB-BL-015).

┼ Systematically check for the 0 value when computing price. This will partly prevent assets from being stolen due to arithmetic rounding (TOB-BL-008, TOB-BL-009, TOB-BL-011).

┼ Favor exact amount-in functions over exact amount-out. This will partly mitigate the effects of arithmetic rounding (TOB-BL-008, TOB-BL-009).

┼ **Review all the documentation related to:**

- How pool limits are handled. This will make sure users are aware of how limits are handled and enforced (TOB-BL-014, TOB-BL-015).

Findings Summary

#	Title	Type	Severity
1	Single-asset liquidity functions allow stealing of assets	Access Controls	High
2	Lack of events in setBLabs is error-prone	Auditing and Logging	Informational
3	Parameters' order of single-asset functions is confusing	Patching	Informational
4	Assets will be lost in case of token migration	Undefined Behavior	Medium
5	Users can silently burn tokens with transfers to 0x0	Data Validation	Low
6	Privileged addresses can be transferred without confirmation even to invalid values	Data Validation	Low
7	Users can join and exit pools even where there are no tokens	Data Validation	Medium
8	Single-asset exit functions allow withdrawing of a negligible amount of assets for free	Data Validation	Undetermined
9	Assets with low decimals or low liquidity lead to withdraw a negligible amount of assets for free	Data Validation	Medium
10	Rounding issues in joinPool/exitPool allow for a negligible amount of free pool tokens	Data Validation	Undetermined
11	Attackers with large funds can steal the pool's assets	Data Validation	High
12	The normalized sum of the weight is not always equal to 1	Undefined Behavior	Informational

13	Pools with a large total supply cause SWAP functions to always revert	Undefined Behavior	Low
14	Token balance limits are declared but not enforced	Undefined Behavior	Informational
15	The swap-in and swap-out ratios are not correctly enforced	Undefined Behavior	Low

1. Single-asset liquidity functions allow stealing of assets

Severity: High
Type: Access Controls
Target: BPool.sol

Difficulty: Low
Finding ID: TOB-BL-001

Description
Incorrect access control in the single-asset liquidity functions allows an attacker to withdraw all the assets.

The Balancer whitepaper states that:

> Controlled pools are configurable by a "controller" address. Only this address can add or remove liquidity to the pool (call join or exit).

However, this requirement is not enforced by the single-asset deposit and withdrawal functions:

- `joinswapExternAmountIn`
- `joinswapExternAmountOut`
- `exitswapPoolAmountIn`
- `exitswapPoolAmountOut`

These functions are callable by anyone as soon as the pool is public, even if it is not finalized. This creates several issues:

1. Anyone can steal the funds.
If `exitswapExternAmountOut` is called on a non-finalized pool, `_totalSupply` is `0`, and `poolAmountIn` will always be `0`:

```
   function exitswapExternAmountOut(address tokenOut, uint tokenAmountOut, uint
maxPoolAmountIn)
        external
        _logs_
        _lock_
        returns (uint poolAmountIn)
    {

        require(_records[tokenOut].bound, "ERR_NOT_BOUND");
        require(_publicSwap, "ERR_SWAP_NOT_PUBLIC");

        Record storage outRecord = _records[tokenOut];

        poolAmountIn = calcPoolInGivenSingleOut(
                            outRecord.balance,
                            outRecord.denorm,
```

```
                _totalSupply,
                _totalWeight,
                tokenAmountOut,
                _swapFee
            );
```

Figure 1.1: `BPool.sol#L652-L671`.

As a result, anyone can call `exitswapExternAmountOut` to withdraw all the assets for free.

2. Liquidity providers will lose their deposits.
Similar to `exitswapExternAmountOut`, `poolAmountOut` will always be 0 in `joinswapExternAmountIn`. As a result, the liquidity providers will never have pool shares and will lose their deposits.

3. Weight can be changed after external deposits
Because the pool is not finalized, the controller can change the asset's weight by calling `rebind`, even after receiving liquidity from external contributors.

Exploit Scenario
Bob creates a pool worth $1,000,000. Bob makes the pool public, but not finalized. Eve steals all the money.

Recommendation
Short term, prevent the single-asset deposit/withdraw functions from being called if the pool is not finalized:

- `joinswapExternAmountIn`
- `joinswapExternAmountOut`
- `exitswapPoolAmountIn`
- `exitswapPoolAmountOut`

Long term, create a state machine representation of the pool, describing each state and each function that should be callable by the different actors.

2. Lack of events in setBLabs is error-prone

Severity: Informational
Type: Auditing and Logging
Target: BFactory.sol

Difficulty: Low
Finding ID: TOB-BL-002

Description
`BFactory.setBLabs` changes the `blabs` address without emitting an event:

```
    function setBLabs(address b) external {
        require(msg.sender == _blabs, "ERR_NOT_BLABS");
        _blabs = b;
    }
```

Figure 2.1: `BFactory.sol#L51-L54.`

The lack of events makes it more difficult to track potential errors or a compromise.

Recommendation
Short term, add an event to `BFactory.setBLabs.`

Long term, monitor the contract's events to detect a potential compromise.

3. Parameters' order of single-asset functions is confusing

Severity: Informational
Type: Patching
Target: BPool.sol

Difficulty: Low
Finding ID: TOB-BL-003

Description
The parameters' order of several similar functions is not the same:

- `joinswapExternAmountIn(address tokenIn, uint tokenAmountIn, uint minPoolAmountOut)`
- `joinswapPoolAmountOut(uint poolAmountOut, address tokenIn, uint maxAmountIn)`
- `exitswapPoolAmountIn(uint poolAmountIn, address tokenOut, uint minAmountOut)`
- `exitswapExternAmountOut(address tokenOut, uint tokenAmountOut, uint maxPoolAmountIn)`

For example, the address of the token is sometimes the first argument, and sometimes the second one. This inconsistency might create confusion for the users.

Recommendation
Short term, follow the same order of parameters for similar functions.

Long term, consider using Slither's printer capacity to review the contract's API.

4. Assets will be lost in case of token migration

Severity: Medium
Type: Undefined Behavior
Target: BPool.sol

Difficulty: High
Finding ID: TOB-BL-004

Description
Once the pool is finalized, no token can be added or removed. If a token is migrated to a new address, the pool will not follow this migration.

In this situation, the pool's value will not stay stable, and all the liquidators are likely to withdraw their deposits as soon as possible, causing the slowest liquidators to lose their deposits.

Exploit Scenario
Alice creates a pool with several tokens, including DAI. Bob deposits 1,000 DAI into the pool. The MakerDAO team decides to migrate to a new version of the token contract to add support for a new feature. Bob wishes to withdraw some of his DAI held in the pool but is unable to as the contract has no functionality to interact with the new DAI contract.

Recommendation
Short term, properly document this limitation.

Long term, Balancer should study which solutions can be implemented, and properly document what happens in the case of migration. Consider that adding a mechanism to handle token migration will significantly increase code complexity, and potentially erode trust in the system. If no on-chain mechanism is present to follow a token's migration, the documentation should highlight off-chain strategies. For example, if the pool's value is significant, it might be possible to contact the token's owner to ask for a migration to a new pool (if all the assets can be migrated).

5. Users can silently burn tokens with transfers to 0x0

Severity: Low
Type: Data Validation
Target: BToken.sol

Difficulty: Low
Finding ID: TOB-BL-005

Description
The amount of minted tokens is tracked in the `BToken` contract by the `totalSupply` function. Burning tokens is only possible using an internal operation (called by `BPools`). However, the `transfer` and `transferFrom` methods do not restrict the address destination of `address(0x0)`, effectively allowing tokens to be burned without decreasing the `totalSupply` variable.

Exploit Scenario
Alice creates a pool and uses some off-chain code to manage it. A calculation results in a transfer to the null or empty address of `0x0`. As a result, Alice loses her tokens.

Recommendation
Add a `require` condition in `transfer` and `transferFrom` that explicitly forbids burning tokens by transferring them to `0x0`.

Long term, use Echidna and Manticore to check that the `BToken` does not easily allow users to burn tokens by transferring them to `0x0`.

6. Privileged addresses can be transferred without confirmation even to invalid values

Severity: Low
Type: Data Validation
Target: BPool.sol, BFactory.sol

Difficulty: Medium
Finding ID: TOB-BL-006

Description
An incorrect use of the functions to set privileged addresses in contracts can irreversibly set them to invalid addresses, such as `0x0`.

The owner or controller of the contracts can change privileged addresses using functions such as `setController` and `setBLabs`:

```
function setController(address manager)
    external
    _logs_
    _lock_
{
    require(msg.sender == _controller, "ERR_NOT_CONTROLLER");
    _controller = manager;
}
```

Figure 6.1: `BPool.sol#L205-L212`.

```
function setBLabs(address b)
    external
{
    require(msg.sender == _blabs,
"ERR_NOT_BLABS");
    emit LOG_BLABS(msg.sender, b);
    _blabs = b;
}
```

Figure 6.2: `BFactory.sol#L63-L69`.

However, these functions do not check for invalid values (e.g., `0x0`), and they work in a single transaction.

Exploit Scenario

Alice creates a pool. She uses some off-chain code to manage it. However, a software issue in her code calls the `setController` function with an uninitialized value (`0x0`). The BPool code accepts this new value and locks up Alice's pool. As a result, she will need to create a new pool.

Recommendation
Short term, split this important functionality into several functions. For instance, to change the current controller, implement `setController` and `acceptController`, which will reject any null address. Additionally, add a `renounceController` function that allows you to set the controller to `0x0` if needed.

Long term, use Echidna and Manticore to verify that the administrative addresses cannot be set to incorrect values.

7. Users can join and exit pools even where there are no tokens

Severity: Medium
Type: Data Validation
Target: BPool.sol

Difficulty: Medium
Finding ID: TOB-BL-007

Description
Incorrect access control in the `joinPool` and `exitPool` functions allow calls to them to succeed even when there are no binded tokens, producing unexpected results.

One one hand, if a user calls `joinPool` when `_tokens.length` is `0`, the pool will produce share tokens for free:

```
    function joinPool(uint poolAmountOut, uint[] calldata maxAmountsIn)
        external
        _logs_
        _lock_
    {
        require(_finalized, "ERR_NOT_FINALIZED");

        uint poolTotal = totalSupply();
        uint ratio = bdiv(poolAmountOut, poolTotal);
        require(ratio != 0, "ERR_MATH_APPROX");

        for (uint i = 0; i < _tokens.length; i++) {
            address t = _tokens[i];
            uint bal = _records[t].balance;
            uint tokenAmountIn = bmul(ratio, bal);
            require(tokenAmountIn <= maxAmountsIn[i], "ERR_LIMIT_IN");
            _records[t].balance = badd(_records[t].balance, tokenAmountIn);
            emit LOG_JOIN(msg.sender, t, tokenAmountIn);
            _pullUnderlying(t, msg.sender, tokenAmountIn);
        }
        _mintPoolShare(poolAmountOut);
        _pushPoolShare(msg.sender, poolAmountOut);
    }
```

Figure 7.1: `BPool.sol#L368-L390`.

On the other hand, if a user calls `exitPool` when `_tokens.length` is `0`, the pool will burn shares, without giving anything in exchange:

```
    function exitPool(uint poolAmountIn, uint[] calldata minAmountsOut)
        external
        _logs_
        _lock_
    {
```

```
        require(_finalized, "ERR_NOT_FINALIZED");

        uint poolTotal = totalSupply();
        uint exitFee = bmul(poolAmountIn, EXIT_FEE);
        uint pAiAfterExitFee = bsub(poolAmountIn, exitFee);
        uint ratio = bdiv(pAiAfterExitFee, poolTotal);
        require(ratio != 0, "ERR_MATH_APPROX");

        _pullPoolShare(msg.sender, poolAmountIn);
        _pushPoolShare(_factory, exitFee);
        _burnPoolShare(pAiAfterExitFee);

        for (uint i = 0; i < _tokens.length; i++) {
            address t = _tokens[i];
            uint bal = _records[t].balance;
            uint tokenAmountOut = bmul(ratio, bal);
            require(tokenAmountOut >= minAmountsOut[i], "ERR_LIMIT_OUT");
            _records[t].balance = bsub(_records[t].balance, tokenAmountOut);
            emit LOG_EXIT(msg.sender, t, tokenAmountOut);
            _pushUnderlying(t, msg.sender, tokenAmountOut);
        }

    }
```

Figure 7.2: `BPool.sol#L392-L419`.

Exploit Scenario

Bob creates a pool and finalizes it without adding any tokens. Then Eve calls `joinPool` and obtains shares for free. Later, she tries to sell her shares to an investor who incorrectly assumes that every pool will have at least one token.

Recommendation

Short term, prevent pool finalization if there are fewer than two tokens.

Long term, use Echidna and Manticore to test that `joinPool` and `exitPool` work as expected.

8. Single-asset exit functions allow withdrawing of a negligible amount of assets for free

Severity: Undetermined
Type: Data Validation
Target: BPool.sol

Difficulty: Low
Finding ID: TOB-BL-008

Description
A rounding issue caused by Solidity's integer arithmetic in `exitswapExternAmountOut` allows users to withdraw assets without burning pool tokens.

The `exitswapExternAmountOut` function computes the amount of pool tokens to be burned with `calcPoolInGivenSingleOut`:

```
   function exitswapExternAmountOut(address tokenOut, uint tokenAmountOut, uint
maxPoolAmountIn)
        external
        _logs_
        _lock_
        returns (uint poolAmountIn)
    {

        require(_records[tokenOut].bound, "ERR_NOT_BOUND");
        require(_publicSwap, "ERR_SWAP_NOT_PUBLIC");

        Record storage outRecord = _records[tokenOut];

        poolAmountIn = calcPoolInGivenSingleOut(
                            outRecord.balance,
                            outRecord.denorm,
                            _totalSupply,
                            _totalWeight,
                            tokenAmountOut,
                            _swapFee
                        );
```

Figure 8.1: `BPool.sol#L652-L671`.

Due to rounding approximation, `calcPoolInGivenSingleOut` can return `0` while `tokenAmountOut` is greater than `0`. As a result, an attacker can withdraw assets without having pool tokens. We've provided Echidna and Manticore scripts that show how to trigger the issue.

Exploit Scenario
Bob has a pool with two assets. The first asset has a balance of 9223372036854775808. There are 9223372036854775808 pool tokens. Eve is able to withdraw 4 wei of the asset for

free. It is worth noting that the amount of tokens that Eve is allowed to obtain for free in a single transaction is bounded by the token precision, which is at least `1/(10**18)`.

Recommendation
Short term, revert in `exitswapExternAmountOut` if `poolAmountIn` is `0`.

Long term, consider:

- Checking for the `0` value when transferring values if appropriate.
- Favoring exact amount-in functions over exact amount-out.
- Using Echidna and Manticore to test the rounding effects.

9. Assets with low decimals or low liquidity lead to withdraw a negligible amount of assets for free

Severity: Medium
Type: Data Validation
Target: BPool.sol

Difficulty: Medium
Finding ID: TOB-BL-009

Description

A rounding issue caused by Solidity's integer arithmetic in `joinPool` allows users to receive free tokens if one of the assets has a low decimal or low liquidity.

The amount of assets to be paid in `joinPool` is:

```
tokenAmountIn = token.balanceOf(this) * (poolAmountOut / poolTotal)
```

```
function joinPool(uint poolAmountOut, uint[] calldata maxAmountsIn)
    external
    _logs_
    _lock_
{
    require(_finalized, "ERR_NOT_FINALIZED");

    uint poolTotal = totalSupply();
    uint ratio = bdiv(poolAmountOut, poolTotal);
    require(ratio != 0, "ERR_MATH_APPROX");

    for (uint i = 0; i < _tokens.length; i++) {
        address t = _tokens[i];
        uint bal = _records[t].balance;
        uint tokenAmountIn = bmul(ratio, bal);
```

Figure 9.1: `BPool.sol#L368-L382`.

The multiplication is done through the fixed-point arithmetic `bmul`:

```
c = ((a * b) + BONE / 2) / BONE
```

```
function bmul(uint a, uint b)
    internal pure
    returns (uint)
{
    uint c0 = a * b;
    require(a == 0 || c0 / a == b, "ERR_MUL_OVERFLOW");
    uint c1 = c0 + (BONE / 2);
    require(c1 >= c0, "ERR_MUL_OVERFLOW");
    uint c2 = c1 / BONE;
```

```
        return c2;
    }
```

Figure 9.2: `BNum.sol#L63-L73`.

If `((a * b) + BONE / 2)` is below `BONE (10**18)`, the result of the multiplication will be `0`.

As a result, if `token.balanceOf(this) * (poolAmountOut / poolTotal) +  < 5 * 10**17, poolAmountOut,` free share tokens will be generated. This will happen if the token has a decimal below 18, or low liquidity.

Some high-value targets have a low decimal. For example, TUSD, which is the first token in terms of market cap (according to etherscan), has a decimal of 6. A similar issue is present in `exitPool`, which can lead a user to burn pool share tokens without receiving any assets back.

Exploit Scenario
Bob creates a pool with the TUSD token. Eve uses the rounding issue to obtain free pool share tokens, and reduce the amount in the pool. It is worth noting that the amount of tokens that Eve is allowed to obtain for free in a single transaction is bounded by the token precision, which is at least `1/(10**18)`.

Recommendation
Short term, revert in `joinPool` if `tokenAmountIn` is zero and `tokenAmountOut` in `exitPool.` Consider preventing tokens with a `decimal != 18` to be bound.

Long term, consider:

- Checking for the `0` value when transferring values, if appropriate.
- Favoring exact amount-in functions over exact amount-out.
- Using Echidna and Manticore to test the rounding effects.

10. Rounding issues in joinPool/exitPool allow for a negligible amount of free pool tokens

Severity: Undetermined
Type: Data Validation
Target: BPool.sol

Difficulty: Medium
Finding ID: TOB-BL-010

Description

Due to rounding issues caused by Solidity's integer arithmetic when depositing and withdrawing an asset, it is possible for an attacker to generate free pool tokens.

When a user asks for `poolAmountOut` pool tokens through `joinPool`, they have to pay `asset.balanceOf(this) * (poolAmountOut / poolTotal)` tokens.

```
    function joinPool(uint poolAmountOut, uint[] calldata maxAmountsIn)
        external
        _logs_
        _lock_
    {
        require(_finalized, "ERR_NOT_FINALIZED");

        uint poolTotal = totalSupply();
        uint ratio = bdiv(poolAmountOut, poolTotal);
        require(ratio != 0, "ERR_MATH_APPROX");

        for (uint i = 0; i < _tokens.length; i++) {
            address t = _tokens[i];
            uint bal = _records[t].balance;
            uint tokenAmountIn = bmul(ratio, bal);
```

Figure 10.1: `BPool.sol#L368-L382`.

When a user exits a pool, they pay `poolAmountIn` pool tokens, and they receive `asset.balanceOf(this) * (poolAmountIn / poolTotal)`.

```
    function exitPool(uint poolAmountIn, uint[] calldata minAmountsOut)
        external
        _logs_
        _lock_
    {
        require(_finalized, "ERR_NOT_FINALIZED");

        uint poolTotal = totalSupply();
        uint exitFee = bmul(poolAmountIn, EXIT_FEE);
```

```
        uint pAiAfterExitFee = bsub(poolAmountIn, exitFee);
        uint ratio = bdiv(pAiAfterExitFee, poolTotal);
        require(ratio != 0, "ERR_MATH_APPROX");

        _pullPoolShare(msg.sender, poolAmountIn);
        _pushPoolShare(_factory, exitFee);
        _burnPoolShare(pAiAfterExitFee);

        for (uint i = 0; i < _tokens.length; i++) {
            address t = _tokens[i];
            uint bal = _records[t].balance;
            uint tokenAmountOut = bmul(ratio, bal);
```

Figure 10.2: `BPool.sol#L392-L412`.

Due to the rounding of these operations, an attacker can find an amount of `poolAmountOut` that will be greater than `poolAmountIn` while allowing the same amount of asset tokens to be transferred. As a result, an attacker can generate free pool tokens by consecutively calling `joinPool` and `exitPool`.

Exploit Scenario
To exploit this issue, the attacker requires:

- EXIT_FEE is equal `0`
- Initial_balance: 4294983682
- Initial pool supply: 2305843009213693953

Then, they need to:

1. call `joinPool` to generate 268435457 pool tokens, and pay 1 wei of the asset
2. call `exitPool` to burn 268434456 pool tokens, and pay 1 wei of the asset

Finally, the attacker receives all the assets, and 1,001 free pool tokens. It is worth noting that the amount of tokens that the attacker is allowed to obtain for free in a single transaction is bounded by the token precision, which is at least `1/(10**18)`.

Recommendation
Fixing this issue requires some code changes. Trail of Bits is still investigating mitigations. One solution could be to compute the dust in `joinPool`, and revert if it is above a threshold.

Long term, consider using Echidna and Manticore to test the rounding effects.

11. Attacker with large funds can steal the pool's assets

Severity: High
Type: Data Validation
Target: BPool.sol

Difficulty: High
Finding ID: TOB-BL-011

Description
A pool with an empty asset balance allows anyone to generate unlimited free share tokens. Such a pool can be emptied by an attacker.

`joinswapPoolAmountOut` is one of the functions to deposit a single asset:

```
   function joinswapPoolAmountOut(uint poolAmountOut, address tokenIn, uint
maxAmountIn)
        external
        _logs_
        _lock_
        returns (uint tokenAmountIn)
    {
        require(_records[tokenIn].bound, "ERR_NOT_BOUND");
        require(_publicSwap, "ERR_SWAP_NOT_PUBLIC");

        Record storage inRecord = _records[tokenIn];

        tokenAmountIn = calcSingleInGivenPoolOut(
                            inRecord.balance,
                            inRecord.denorm,
                            _totalSupply,
                            _totalWeight,
                            poolAmountOut,
                            _swapFee
                        );

        require(tokenAmountIn <= maxAmountIn, "ERR_LIMIT_IN");

        inRecord.balance = badd(inRecord.balance, tokenAmountIn);
```

Figure 11.1: `BPool.sol#L582-L604`.

If `inRecord.balance` is `0`, `calcSingleInGivenPoolOut` will return `0`:

```

/**********************************************************************************
************
    // calcSingleInGivenPoolOut
//
    // tAi = tokenAmountIn              //(pS + pAo)\     /    1    \\
//
```

```
    // pS = poolSupply                 || ---------  | ^ | --------- || * bI - bI
//
    // pAo = poolAmountOut              \\    pS    /     \(wI / tW)//
//
    // bI = balanceIn          tAi =  ----------------------------------------
//
    // wI = weightIn                              /      wI  \
//
    // tW = totalWeight                          |  1 - ----  |  * sF
//
    // sF = swapFee                               \      tW  /
//

**********************************************************************************
***********/
    function calcSingleInGivenPoolOut(
        uint tokenBalanceIn,
        uint tokenWeightIn,
        uint poolSupply,
        uint totalWeight,
        uint poolAmountOut,
        uint swapFee
    )
        public pure
        returns (uint tokenAmountIn)
    {
        uint normalizedWeight = bdiv(tokenWeightIn, totalWeight);
        uint newPoolSupply = badd(poolSupply, poolAmountOut);
        uint poolRatio = bdiv(newPoolSupply, poolSupply);

        //uint newBalTi = poolRatio^(1/weightTi) * balTi;
        uint boo = bdiv(BONE, normalizedWeight);
        uint tokenInRatio = bpow(poolRatio, boo);
        uint newTokenBalanceIn = bmul(tokenInRatio, tokenBalanceIn);
        uint tokenAmountInAfterFee = bsub(newTokenBalanceIn, tokenBalanceIn);
        // Do reverse order of fees charged in joinswap_ExternAmountIn, this way
        //     ``` pAo == joinswap_ExternAmountIn(Ti, joinswap_PoolAmountOut(pAo,
Ti)) ```
        //uint tAi = tAiAfterFee / (1 - (1-weightTi) * swapFee) ;
        uint zar = bmul(bsub(BONE, normalizedWeight), swapFee);
        tokenAmountIn = bdiv(tokenAmountInAfterFee, bsub(BONE, zar));
        return tokenAmountIn;
    }
```

Figure 11.2: `BMath.sol#L147-L183`.

As a result, depositing assets in a pool with an empty balance generates free pool tokens. An attacker with enough funds can empty any pool of assets. Pools with low liquidity or assets with low decimals are more likely to be vulnerable.

Exploit Scenario

Bob has a pool with $10,000 of TUSD (6 decimals) and $10,000 of DAI (18 decimals). Eve has $10,000,000. Eve buys all the TUSD from the pool, generates free pool tokens, and empties both assets from the pool. Altogether, Eve steals $20,000.

Recommendation
Short term, revert in `joinswapPoolAmountOut` if `calcSingleInGivenPoolOut` is zero.

Long term, check for the `0` value when transferring values, if appropriate. Use Echidna and Manticore to test the rounding effects.

12. The normalized sum of the weight is not always equal to 1

Severity: Informational
Type: Undefined Behavior
Target: BPool.sol

Difficulty: Low
Finding ID: TOB-BL-012

Description
The normalized sum of the token weight may not always be equal to 1. The Balancer whitepaper states that the sum of normalized token weights should be equal to 1. However, inherent rounding issues in the division performed by the `getNormalizedWeight` function can accumulate in the sum and result in a value less than 1.

Exploit Scenario
Alice creates a new pool. She reviews the documentation and incorrectly assumes that the sum of a normalized weight token will be 1. As a result, she incorrectly implements on-chain/off-chain code to interact with the pool, potentially causing unexpected results (e.g., rounding issues, zero division) in her code.

Recommendation
Short term, properly document this rounding issue and make sure users understand that this property is not strictly enforced.

Long term, consider using Echidna and Manticore to ensure normalized weights are correctly computed.

13. Pools with a large total supply cause SWAP functions to always revert

Severity: Low
Type: Undefined Behavior
Target: BPool.sol

Difficulty: High
Finding ID: TOB-BL-013

Description
A revert that occurs during the computations performed of SWAP functions can stop users from calling these functions, if the initial supply of tokens is large.

When a pool is finalized, the initial supply of shares is created. There is a lower bound to the initial supply, but no upper bound:

```
    function finalize(uint initSupply)
        external
        _logs_
        _lock_
    {
        require(msg.sender == _controller, "ERR_NOT_CONTROLLER");
        require(!_finalized, "ERR_IS_FINALIZED");
        require(initSupply >= MIN_POOL_SUPPLY, "ERR_MIN_POOL_SUPPLY");

        _finalized = true;
        _publicSwap = true;

        _mintPoolShare(initSupply);
        _pushPoolShare(msg.sender, initSupply);
    }
```

Figure 13.1: `BPool.sol#L224-L238`.

The total supply is used in several places, i.e., in the `joinswapExternAmountIn` function, which calls `calcPoolOutGivenSingleIn`:

```
uint newPoolSupply = bmul(poolRatio, poolSupply);
```

Figure 13.2: `BMath.sol#L142`.

The multiplication is done through the fixed-point arithmetic `bmul`:

```
    function bmul(uint a, uint b)
        internal pure
        returns (uint)
    {
        uint c0 = a * b;
        require(a == 0 || c0 / a == b, "ERR_MUL_OVERFLOW");
```

```
        uint c1 = c0 + (BONE / 2);
        require(c1 >= c0, "ERR_MUL_OVERFLOW");
        uint c2 = c1 / BONE;
        return c2;
    }
```

Figure 13.3: `BNum.sol#L63-L73`.

An overflow in this computation will revert, regardless of the input values used in the SWAP functions.

Exploit Scenario
Bob creates a pool with a very large number of token shares. Alice tries to call a SWAP function, but it only reverts, regardless of the input values she uses. As a result, she is unable to use the pool as expected.

Recommendation
Short term, document this behavior and make sure the users are aware of it.

Long term, consider using Echidna and Manticore to detect this kind of issues in the codebase.

14. Token balance limits are declared but not enforced

Severity: Informational
Type: Undefined Behavior
Target: BPool documentation

Difficulty: Low
Finding ID: TOB-BL-014

Description
Although the documentation states that there are maximum and minimum values for token balances, there is no code to enforce such limits.

The documentation states:

```
Minimum Balance - (10^18) / (10^12)
The minimum balance of any token in a pool is 10^6 wei. Important note: this
is agnostic to token decimals and may cause issues for tokens with less than
6 decimals.

Maximum Balance - (10^18) * (10^12)
The maximum balance of any token in a pool is 10^12 ether.
```

However, it is still possible to have a token balance larger than the maximum or smaller than the minimum, using `joinPool` and `exitPool`, respectively.

Exploit Scenario
Alice creates a new pool. She reviews the documentation and incorrectly assumes that the token balances are bounded and the limits are correctly enforced. As a result, she incorrectly implements on-chain/off-chain code to interact with the pool, potentially causing unexpected results (e.g., rounding issues, zero division) in her code.

Recommendation
Short term, properly document the maximum and minimum values for token balances to make sure users understand that these are not enforced. It is worth mentioning that enforcing the limits in the contract could open the door for new denial-of-service attacks.

Long term, review all the documentation regarding pool limits. Consider using Echidna and Manticore to test that pool limits are always enforced.

15. The swap-in and swap-out ratios are not correctly enforced

Severity: Low
Type: Undefined Behavior
Target: BPool.sol

Difficulty: Low
Finding ID: TOB-BL-015

Description
The limits on the ratios to swap in and swap out tokens are not always correctly enforced. The Balancer documentation defines maximum ratios when performing swap-in and swap-out operations:

```
Maximum Swap In Ratio - 1/2
A maximum swap in ratio of 0.50 means an user can only swap in less than 50%
of the current balance of tokenIn for a given pool

Maximum Swap Out Ratio - 1/3
A maximum swap out ratio of 1/3 means an user can only swap out less than
33.33% of the current balance of tokenOut for a given pool
```

To define these limits, there are two constants in the `BConst` contract:

```
    uint public constant MAX_IN_RATIO      = BONE / 2;
    uint public constant MAX_OUT_RATIO     = (BONE / 3) + 1 wei;
```

Figure 15.1: `BConst.sol#L39-L40`.

These limits are supposed to be enforced in `swapExactAmountIn` and `swapExactAmountOut`:

```
require(tokenAmountIn <= bmul(inRecord.balance, MAX_IN_RATIO), "ERR_MAX_IN_RATIO");
```

Figure 15.2: `BPool.sol#L442&`.

```
require(tokenAmountOut <= bmul(outRecord.balance, MAX_OUT_RATIO),
"ERR_MAX_OUT_RATIO");
```

Figure 15.3: `BPool.sol#L504`.

However, it still seems to be possible to swap over the limits, since the checks are performed directly using the token balance. These values are taken directly from the token supplies and the constants are made using `BONE`, so the result is not as precise as expected.

Exploit Scenario
Bob creates a new pool, and several users join. They review the documentation and note the swap limits. However, Eve is able to swap tokens over the limits. The users observe the

large swaps from Eve so they decide to exit the pool since they believe their funds are no longer secure.

Recommendation
Short term, simplify the implementation of the ratio checks in the swap-in and swap-out operations using `bdiv`.

Long term, review all the documentation regarding pool limits. Consider using Echidna and Manticore to test that pool limits are always enforced.

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 system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or 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 user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

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

B. Fix Log

Balancer Labs addressed issues TOB-BL-001 to TOB-BL-015 in their codebase as a result of the assessment. Each of the fixes was verified by Trail of Bits. The reviewed code is available in git revision `e232d03eea1c66529f22d3157c7f560bf0782370`.

ID	Title	Severity	Status
01	Single-asset liquidity functions allow stealing of assets	High	Fixed
02	Lack of events in setBLabs is error-prone	Informational	Fixed
03	Parameters' order of single-asset functions is confusing	Informational	Fixed
04	Assets will be lost in case of token migration	Medium	Mitigated
05	Users can silently burn tokens with transfers to `0x0`	Low	Won't fix
06	Privileged addresses can be transferred without confirmation even to invalid values	Low	Won't fix
07	Users can join and exit pools even where there are no tokens	Medium	Fixed
08	Single-asset exit functions allow withdrawing of a negligible amount of assets for free	Undetermined	Mitigated
09	Assets with low decimals or low liquidity lead to withdraw a negligible amount of assets for free	Medium	Mitigated
10	Rounding issues in joinPool/exitPool allow for a negligible amount of free pool tokens	Undetermined	Mitigated
11	Attacker with large funds can steal the pool's assets	High	Fixed
12	The normalized sum of the weight is not always equal to 1	Informational	Fixed
13	Pools with a large total supply cause SWAP functions to always revert	Low	Mitigated
14	Token balance limits are declared but not enforced	Informational	Fixed
15	The swap-in and swap-out ratios are not correctly enforced	Informational	Won't fix

Detailed Fix Log

This section includes brief descriptions of fixes implemented by Balancer after the end of this assessment that were reviewed by Trail of Bits.

Finding 1: Single-asset liquidity functions allow stealing of assets
This appears to be resolved forcing the pool to be finalized before calling the affected functions.

Finding 2: Lack of events in setBLabs is error-prone
This appears to be resolved by adding an event in `setBLabs`.

Finding 3: Parameters' order of single-asset functions is confusing
This appears to be resolved by reordering the parameters of the affected functions.

Finding 4: Assets will be lost in case of token migration
This appears to be partly resolved by adding a warning about token migration in the documentation. However, we recommend studying different strategies, as well as their costs and limitations to expand the documentation.

Finding 5: Users can silently burn tokens with transfers to `0x0`
The Balancer Labs team indicated that they will not fix the issue because they say no restrictions to `0x0` addresses will be added to the core protocol.

Finding 6: Privileged addresses can be transferred without confirmation even to invalid values
The Balancer Labs team indicated that they will not fix the issue, saying that `setBLabs` will not be used in the bronze release since the EXIT_FEE is `0` and the additional UX complexity does not outweigh the benefits of splitting `setController`.

Finding 7: Users can join and exit pools even where there are no tokens
This appears to be resolved by forcing pool finalization to have two or more tokens binded.

Finding 8: Single-asset exit functions allow withdrawing of a negligible amount of assets for free
This appears to be mitigated by adding a check that prevents division errors from reaching zero, and fixing the initial pool share to 100. While it is still technically possible to take advantage of rounding errors increasing the pool shares, it incurs an extremely high cost for the attacker. The Balancer Labs team has committed to monitor pools individually and warn users if a potential attack could happen.

Finding 9: Assets with low decimals or low liquidity lead to withdraw a negligible amount of assets for free
This appears to be mitigated by adding a check that prevents division errors from reaching zero, and fixing the initial pool share to 100. While it is still technically possible to take advantage of rounding errors increasing the pool shares, it incurs an extremely high cost for the attacker. The Balancer Labs team has committed to monitor pools individually and warn users if a potential attack could happen.

Finding 10: Rounding issues in joinPool/exitPool allow for a negligible amount of free pool tokens
This appears to be mitigated by fixing the initial pool share to 100. While it is still technically possible to take advantage of rounding errors increasing the pool shares, it incurs an extremely high cost for the attacker. The Balancer Labs team has committed to monitor pools individually and warn users if a potential attack could happen.

Finding 11: Attacker with large funds can steal the pool's assets
This appears to be fixed adding a check to avoid division errors to reach zero by and using minimum and maximum ratios on swap functions.

Finding 12: The normalized sum of the weight is not always equal to 1
This appears to be resolved by properly documenting this behavior.

Finding 13: Pools with a large total supply cause SWAP functions to always revert
This appears to be mitigated by fixing the initial pool share to 100. While it is still technically possible to take advantage of rounding errors increasing the pool shares, it incurs an extremely high cost for the attacker, requiring to increase the current balance several orders of magnitude. The Balancer Labs team has committed to monitor pools individually and warn users if a potential attack could happen.

Finding 14: Token balance limits are declared but not enforced
This appears to be resolved by removing the maximum balance limit and clarifying how the minimum balance limit is enforced in the documentation.

Finding 15: The swap-in and swap-out ratios are not correctly enforced
The Balancer Labs team indicated that they will not fix the issue, indicating that the current implementation already works as expected, considering the limitations in the integer arithmetic imposed by Solidity.