CERTORA

Issues found for Balancer V2

Summary

This document describes the specification and verification of **Balancer** using Certora Prover. The work was undertaken from **February 15th to April 19th**. The latest commit that was reviewed and run through the Certora Prover was **47825b3b6a821e18991e193101dbfb8c435f0959**.

The scope of our verification was the **Vault** contract.

The Certora Prover proved the implementation of the **Vault** is correct with respect to the formal rules written by the **Balancer** and the Certora teams. During the verification process, the Certora Prover discovered various issues in the code, some major. All issues were corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is constantly working to remedy these limitations. In this document, we describe the most interesting issues that we found in the Vault contract. The formal descriptions of the specifications we used to find these issues will be made available soon after the publication of this report at: https://www.certora.com/pubs/BalancerApr2021.pdf.

Discovered Issues

Issue	Providing the same token address multiple times in the flash loan's input token address array causes tax evasion and insolvency.
Severity	**High**
Rule(s) broken	Flash loan additivity
Description	Instead of calling a flash loan on token T for amount K with [T], [K] as input parameters, calling the flash loan with [T, T, ...] N times and amounts [N/K, N/K, ...] N times, the user only pays 1/N of the full flash loan fee. The system still thinks it collected the entire fee, making it insolvent. Consequently, the vault owner can withdraw money that was

	incorrectly counted as collected fees at the expense of users. The effects of this attack can be exponentially increased. $\forall$ i, amounts[i+1] = $\frac{1}{1+1/f}$ * amounts[i]. The total fees we pay this way is $\frac{1}{(1+1/f)^n - 1}$, where *f* is the flash loan fee ratio (a number between 0 and 0.01) and *n* is the length of the tokens array. Using this scheme, we decrease the fees exponentially by *n*. For example, with an array of length ten and a maximal fee of 1%, we only need to pay a share of $9*10^{-21}$ of the amount loaned.
Fix	The token input array must now be sorted in strictly ascending order. This prevents using the same token address more than once at the input array.

Issue	Funds invested in pools are locked when the system is paused
Severity	**High**
Rule(s) broken	The integrity of the emergency recovery scheme
Description	When the system is paused, users should take their funds out of the system: withdraw everything from their internal balance and exit all pools they ever joined. Calling *exitPool()* always reverted during a pause. This happened because of a call to *_joinOrExit()*, a shared function for joins and exits with a *TemporarilyPausable* modifier, that prevents transactions temporarily. There was no way to retrieve user funds stored inside pools during a pause, severely undermining the emergency recovery scheme.
Fix	Move the TemporarilyPausable modifier from the shared *_joinOrExit()* function to the external function *joinPool()*.

Issue	No withdrawal tax was charged when users withdrew funds from their internal balance if they swapped with a pool that block.
Severity	**Medium**
Rule(s) broken	Withdrawal tax evasion
Description	There was a mechanism in place to exempt the withdrawal fee if a user used their internal balance temporarily. The intended tax-free scenario was when a user deposited money to their internal balance, used it in a swap, and withdrew it, all during the same block. However, the mechanism also exempted withdrawals if a swap happened during the same block, regardless of when the funds were deposited. Before this mechanism was introduced, a user could still avoid paying the withdrawal fee by constructing a custom pool with a backdoor, skipping the withdrawal step entirely. The solution to that problem was to set the withdrawal fee so low that it would not be worth the hassle. With this mechanism, even honest users will evade paying the withdrawal fee, and the effort required for intentional tax evasion is minimal.
Fix	Withdrawal fees are no longer charged.

Issue	Wrong error reported when an unregistered pool's id is given to a swap function.
Severity	**Low**
Problem	Wrong error message
Description	When a pool id for an unregistered pool was passed to a swap, we reverted with the "TOKEN NOT REGISTERED" error. All pool specializations share this behavior.

Fix	A check if the pool is registered was added. If it is not, we revert with the error “POOL_NOT_REGISTERED”.

Issue	Obscure revert with no reason string for most illegal pool ids in swaps.
Severity	**Low**
Problem	Wrong error message
Description	When swapping with a pool, the pool’s specialization was checked before checking whether that pool was registered. The pool’s id contains two specialization bytes, but the specialization enum has only three values. If the calculated specialization fell out of the enum range, we would revert without a reason string.
Fix	An explicit check of whether the calculated pool specialization is in the legal range was added. If it is not, we revert with the error “INVALID_POOL_ID”.

Issue	No bulk role revoke functions.
Severity	**Low**
Problem	Compromised addresses recovery scheme.
Description	An address can be granted many roles, say by the *grantRoles()* or *grantRolesToMany()* functions. If that address is later compromised, we want to have an easy and efficient way to revoke all of its roles. Reverse operations to *grantRoles()* and *grantRolesToMany()* did not exist; roles could be revoked one at a time, individually.
Fix	Bulk revoke functions *revokeRoles()* and *revokeRolesFromMany()* were added.

Issue	*TokenRegistered* event emitted does not reflect possible reordering of tokens of a TwoTokensPool.
Severity	**Low**
Rule(s) broken	The order of tokens in a pool is constant for all operations besides *deregisterTokens()*.
Description	A user may register any two different non-zero token addresses to a *TwoTokensPool*. The emitted event will report the token addresses in the order the user inserted them. When a user gives two descending token addresses as input, the vault will sort them in ascending order. The vault requires the token addresses to be sorted in ascending order for any subsequent action. However, the emitted event does not indicate the order change to the user.
Fix	Require the token addresses to be input in (strictly) ascending order.

Issue	No reverts when there are more asset managers than tokens in registerTokens().
Severity	**Low**
property broken	Each registered token must have an asset manager.
Description	At PoolRegistry.sol in *registerTokens()*, we did not verify that the *tokens* and *assetManagers* input arrays had equal lengths. If the array of tokens was longer, we reverted to an out-of-bounds exception. If there were more managers than tokens, the last reminding managers in the input array were ignored.
Fix	Add `InputHelpers.ensureInputLengthMatch(tokens.length, assetManagers.length)` to *registerTokens()*.

Issue	Wrong error message when the first swap in a batch swap consisting of two or more swaps had an amount of zero.
Severity	**Low**
Problem	Wrong error message, possible optimization
Description	If the first swap in a batch swap consisting of two or more swaps had an amount of zero, it reverted with an error message “MALCONSTRUCTED_MULTIHOP_SWAP”.
Fix	The if condition changed from (*swaps.length* > *1*) to (*i* > *0*). Now, if the first token has an amount of zero, we revert with the error “UNKNOWN_AMOUNT_IN_FIRST_SWAP”. We also fail faster in this case, reducing gas costs.

Issue	Wrong error message when the first token given to a flash loan is the zero address token.
Severity	**Low**
Problem	Wrong error message.
Description	When the first token in the input token array of a flash loan was the zero token, we reverted with the error “UNORDERED_TOKENS”. That happened even if the tokens were ordered, and even if there was only a single token in the input array.
Fix	An explicit check of whether a token is the zero token or not was added. A helper function was added to check if the array is sorted. The helper function ignores arrays of length less than two.

Issue	Inconsistent error messages between *WeightedPool* and *StablePool*.
Severity	**Low**
Problem	Inconsistent error messages
Description	Different error messages were reported by different pool types when

	we insert two input arrays of unequal length in the functions *_onInitializePool* and *_joinExactTokensInForBPTOut*. *StablePool* used a helper library function, while *WeightedPool* implemented the check and returned a different error message.
Fix	The helper function was used in *WeightedPool* as well for consistency.

Issue	Array out of bounds error when joining or exiting a *TwoTokensPool* with no registered tokens.
Severity	**Low**
Problem	Inconsistent behavior between pool types; bad error message
Description	When we joined or exited a *TwoTokensPool*, the code referenced indices 0 and 1 in the *tokens* input array. However, a *TwoTokensPool* might not have any registered tokens. When a user tried to join or exit a *TwoTokensPool* with no registered tokens and provided an empty array as an input, we did not fail the token correctness check for the pool. However, we referenced illegal indices and reverted with an array out of bounds error. This is not just a confusing error message for the user; it was also a different behavior than pools of type *MinimalSwapInfoPool* and *GeneralPool*, which did not fail in this scenario. Note that joining or exiting pools with no registered tokens should not affect the system.
Fix	As a part of the token validation function, we also ensure that the pool has at least one registered token and report an appropriate error message if it does not.

Issue	Trivial *require* statements.
Severity	**Recommendation**
Problem	Wasted resources - gas, bytecode size
Description	Two *require* statements in BalancerPoolToken.sol could never fail because of the if condition they were inside. These statements had no functional effect on the code but increased the contract's byte code, run time, and gas costs.
Fix	The trivial *require* statements were removed.

Issue	Unused variable
Severity	**Recommendation**
Problem	Hard to read code
Description	In InternalBalance.sol, we declared a variable *amountToSend* as an alias to the variable amount. However, the variable amount was never used.
Fix	The variable was renamed as the alias name.

Issue	Unused interface
Severity	**Recommendation**
Problem	Wasted resources - bytecode
Description	No contract implemented the interface *IAuthorizer*
Fix	The *Authorizer* contract now implements the *IAuthorizer* interface.

Issue	Unused functions in helper libraries
Severity	**Recommendation**
Problem	Wasted resources - bytecode, code readability
Description	There are several libraries in the project which are based on OpenZeppelin libraries. They were altered to include unique error codes and other optimizations. However, most files included functions and structs that were never used by any project file. The relevant files were: AccessControl.sol, Address.sol, EnumerableSet.sol, Context.sol, Counters.sol, and SafeCast.sol
Fix	All unused functions and structs were removed.

Issue	Unused file
Severity	**Recommendation**
Problem	Wasted resources - bytecode, code readability
Description	The file contracts/lib/openzeppelin/EnumerableMap.sol was never used. contracts/lib/helpers/EnumerableMap.sol was used instead.
Fix	The unused file contracts/lib/openzeppelin/EnumerableMap.sol was deleted.

Issue	Unused inheritances
Severity	**Recommendation**
Problem	Error prevention
Description	The *Fees* contract inherited from *ReentrancyGuard* and *VaultAuthorization* despite not using any of their functions or variables.

	The *InternalBalance* contract inherited from *Fees* but could inherit from *VaultAuthorization* only.
Fix	Unused inheritances were removed.

Issue	Reentrancy to modifiers
Severity	**Recommendation**
Problem	Error prevention
Description	There are several function modifiers in the code. One of them is *nonReentrant*, which prevents calling the same code twice (say, from within a flash loan). However, if we do not list *nonReentrant* as the first modifier, reentrancy is possible to the modifiers' code that appears before it. All modifiers were reentrancy safe, but this could potentially prevent future issues at no cost.
Fix	The *nonReentrant* modifier is now always the first modifier for all functions.

Issue	Duplicated statements with no effect
Severity	**Recommendation**
Problem	Wasted resources - bytecode size, run time, gas; readability
Description	In the file PoolAssets.sol, there were two duplicate if-else statements. They had no functional change but increased the bytecode size of the contract, as well as the run time and hence gas costs.
Fix	The duplicate statements were removed.