Bera Bex Security Review

Auditors

0xIcingdeath, Lead Security Researcher

Xiaoming90, Security Researcher

J4X, Security Researcher

Report prepared by: Lucas Goiriz

November 7, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Berachain is an EVM-identical L1 turning liquidity into security powered by Proof Of Liquidity.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of balancer-v2-monorepo according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 3 weeks days in total, Berachain engaged with Spearbit to review the balancer-v2-monorepo protocol. In this period of time a total of **10** issues were found.

Summary

Project Name	Berachain
Repository	balancer-v2-monorepo
Commit	5b9e1b...eed5
Type of Project	DEX, DeFi
Audit Timeline	Sep 27 to Oct 23

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	4	2	2
Gas Optimizations	0	0	0
Informational	6	4	2
Total	**10**	**6**	**4**

5 Findings

5.1 Low Risk

5.1.1 Risks of the Balancer protocol should be explicitly documented

Severity: Low Risk

Context: Global scope

Description: Due to the high level of complexity that the Vault brings to the system, we recommend documenting all historical issues that has affected Balancer, to ensure that these issues are adequately mitigated in this Balancer Fork. An example of such analysis can be seen below.

Issue	Affected Pool
Rate Manipulation (exploited) - Aug 2023	Boosted Pools, Linear Pools
Rate Manipulation < 1 - June 2023	Boosted Pools, Composable Stable Pools
Read-only reentrancy - Feb 2023	Stable Pool, Phantom Stable, Linear Pool, Composable Stable, Weighted, Mar
Incorrect Rounding – Oct 2023	Boosted Pool, Linear, Composable Stable Pool
Merkle Orchard Duplicate Claims – Feb 2023	Merkle Orchard, Vault
Privilege Escalation – Dec 2022	`AuthorizerAdapter`
Double entry point tokens – May 2022	Vault, ProtocolFeeCollector
Modification of pool parameters – May 2022	Stable Pool Factory, Meta Stable Pool Factory, Stable Phantom Pool, Investme

Once Berachain commences discusses with Balancer Integrations team, the timelock/authorizer contract flows should be double-checked.

Recommendation: Document all the known issues of the Balancer codebase, including their mitigations and if Berachain's mitigation differs, how. This should include in-code documentation for all mitigations, as well.

Berachain: Acknowledges the risk.

Spearbit: Client acknowledges the risks.

5.1.2 Missing `address(0)` check in `setPOLFeeCollector()`

Severity: Low Risk

Context: ProtocolFeesCollector.sol#L98

Description: The `setPOLFeeCollector()` can be used to set the `polFeeCollector` variable.

```
function setPOLFeeCollector(address _polFeeCollector) external override authenticate {
    polFeeCollector = _polFeeCollector;
    emit POLFeeCollectorChanged(_polFeeCollector);
}
```

Unfortunately, this function never verifies that the variable is set to an actual address and was not accidentally called with zero parameters.

Recommendation: We recommend checking that `_polFeeCollector` is not zero.

```
function setPOLFeeCollector(address _polFeeCollector) external override authenticate {
    require(_polFeeCollector != address(0), "polFeeCollector can not be set to zero");
    polFeeCollector = _polFeeCollector;
    emit POLFeeCollectorChanged(_polFeeCollector);
}
```

Berachain: Fixed in PR 6.

Spearbit: Fixed in the PR provided by the protocol.

5.1.3 Missing check for `polFeeCollector` initialization

Severity: Low Risk

Context: ProtocolFeesCollector.sol#L98

Description: The `withdrawCollectedFeesToPOLFeeCollector()` function allows to withdraw fees to the `polFeeCollector` address. The constructor does not set this address to keep the changes as small as possible. As a result, the address needs to be set via an external setter by an admin.

```
function setPOLFeeCollector(address _polFeeCollector) external override authenticate {
    polFeeCollector = _polFeeCollector;
    emit POLFeeCollectorChanged(_polFeeCollector);
}
```

If an admin forgets to set the address and calls `withdrawCollectedFeesToPOLFeeCollector()` before, the funds will be burned/sent to the zero address.

Recommendation: The issue can be mitigated by adding a check that verifies that the `polFeeCollector` is set to a different address than `address(0)`.

```
/// @notice Function to transfer fees to POL fee collector
function withdrawCollectedFeesToPOLFeeCollector(IERC20[] calldata tokens, uint256[] calldata amounts)
    external
    override
    nonReentrant
    authenticate
{
    InputHelpers.ensureInputLengthMatch(tokens.length, amounts.length);

    require(polFeeCollector != address(0), "polFeeCollector not set");

    for (uint256 i = 0; i < tokens.length; ++i) {
        IERC20 token = tokens[i];
        uint256 amount = amounts[i];
        token.safeTransfer(polFeeCollector, amount);
    }
}
```

Berachain: Fixed by checking that the `POLFeeCollector` can't be set to 0 in the constructor in PR 6.

Spearbit: Mitigated by the fix provided by the protocol.

5.1.4 Bundled creation could be frontrunned

Severity: Low Risk

Context: PoolCreationHelper.sol#61, PoolCreationHelper.sol#L92

Description: A user can use the `PoolCreationHelper` to bundle the creation and initialization of a pool. This is done by calling `factory.create()` and then `vault.joinPool()` in a single transaction. Unfortunately, this bundling can be DoSd by anyone as the factory is also directly accessible. As the `msg.sender` is never used, a malicious actor could front run a call to a bundled TX and create a pool with the same parameters as the user. In that case, the user's call to `factory.create()` would fail, as a pool with the same values and salt was already deployed.

As a result, bundling can permanently be blocked, and users can not prevent themselves from getting DoSd in this scenario.

Recommendation: The issue can't be mitigated in code on the Berachain side, as any changes to the salt could still be predicted by the attacker. Nevertheless we recommend adding documentation for this case, as effectively all usage of the helper can be blocked by an attacker.

Berachain: Acknowledged.

Spearbit: Acknowledged by the team.

5.2 Informational

5.2.1 Expand Balancer's default 3 month pause window on factories

Severity: Informational

Context: Global scope

Description: By default, Balancer's V5 pools implements a 90 day period, in which the pools can be paused. This was implemented early on in deployment, and would have protected Balancer from the loss of funds in future vulnerability disclosures over the past few years.

Balancer V6 extends this to a 4 year pause window and a 6 month buffer period, since a longer period would allow them to pause a pool if needed quickly, and not have to rely on users removing their funds, which is what Balancer needed to do for a lot of the vulnerabilities targeting the vault and the individual Pools. Due to the complexity of this system, bugs aren't quick to find nor easy, thus the 3 month window is an in general, insufficient amount of time to determine if there are actually no bugs for this contract to run indefinitely.

> Estimating the pause window
>
> After pouring our heart and soul into it for nearly a year (not to mention three full-scale audits), a high degree of confidence in our code is understandable, but for the V2 launch -- and all subsequent releases -- we went full Elton John on the rose-colored glasses.
>
> In hindsight, a three-month pause window seems hopelessly naive. We discovered the first vulnerability in the Vault two years after launch. Similarly, the Linear Pool was launched in December '21: and the critical vulnerability was overlooked for nearly as long.
>
> We also launched without Recovery Mode (though, to be fair, we didn't really need it when all we had were V1 Weighted Pools). Not anticipating the complexity of the pool types that would follow, we realized (while addressing an unrelated issue) that our original "proportional exit" escape hatch was insufficient.
>
> If the pause window for Linear Pools had been two or three years instead of three months, the mitigation would have been trivial: simply pause all Linear Pools. As it was, even the L2 pause windows had all expired. We could only pause the V5 Composable Stable Pools, ironically due to previous vulnerabilities that prompted migration from older pools.
>
> It was pure blind luck that the yield token wrappers were upgradeable. Had they been immutable, we would only have been able to mitigate a small minority of pools (the V5 Stables). And if we had not had

Recovery Mode, the mitigation would likewise have been impossible (i.e., the intervention would have also blocked withdrawals).

A truly "infinite" pause window would compromise our permissionless status -- the original reason for limiting it -- but it seems reasonable to choose a pause window to roughly correspond to the expected life of a major protocol version (e.g., 3--5 years).

See the Balancer Post-Mortem on rate manipulation exploit for more details.

As deployment scripts are not in scope for this review, we recommend ensuring that the deployment of the factories include the longer pause windows and buffer period durations, and match the behaviour of V6.

Recommendation: Ensure to document the differences between V5 and V6 for the pools, and ensure that the system is deployed in the same ways.

Berachain: Berachain acknowledged and will make appropriate changes in the deployment script.

Spearbit: Client acknowledges the risk.

5.2.2 Use of indefinite `joinPool` role for callers

Severity: Informational

Context: `PoolCreationHelper`

Description: To authorize a retailer, one needs to call `Vault.setRelayerApproval` and on the protocol level, one needs to call `Vault.grantPermission`. The `setRelayerApproval` can be set indefinitely or for a one-time approval. The assumption in this system is that each caller provides the CreationHelper indefinite relayer approval for the `joinPool` `VaultAction`. In the future, if the `CreationHelper` calls the vault for more than just the `joinPool`, the system should consider using a one-time approval to limit the effects of what a relayer could do to a users' funds while transacting with the protocol.

```
// Grant PoolCreationHelper to call Vault.joinPool
const joinPoolRole = await actionId(vault.instance, 'joinPool');
await vault.grantPermissionGlobally(joinPoolRole, poolCreationHelper.address);
// ...
// Approve poolCreationHelper as relayer for the caller
await vault.setRelayerApproval(caller, poolCreationHelper.address, true);
```

Recommendation: Document the system currently assuming infinite relayer approval on a user level for joinPool only. Either in the code or in developer documentation, make sure to document the impacts of adding other calls to such helper contracts, and ensure that they use one-time approvals.

Berachain: Given current helper contract is non-upgradable, other `vaultAction` than `joinPool` can't be supported hence we don't see any impact of indefinite relayer approval.

Spearbit: Client acknowledges the risk.

5.2.3 Non-standard token compatibility needs to be documented

Severity: Informational

Context: Vault.sol

Description: Out of the box, Balancer's Vault does not support non-standard tokens, including:

- Tokens with fee: results in the accounting of the vault being incorrect.
- Rebasing tokens: results in the accounting being off due to sudden increases or decreases.
- Double entry point tokens: similar to previous protocol fee related issues with Balancer, the system is not designed to support such tokens.
- Tokens with non-standard decimals (>18, or no decimals field): will revert on creation, as accounting assumes a max of 18 decimals.

Recommendation: Document the limitations of the Vault in Berachain documentation and the `PoolCreationHelper` contract, so users creating pools know the limitations of these pools and how they should be used. On the Berachain UI, ensures that the risks associated to these tokens are noted.

Berachain: Documented on the `PoolCreationHelper.sol` in PR 6 contract.

Spearbit: Berachain documented the non-standard token compatibility. User documentation should eventually be updated as well.

5.2.4 Initial balance for BPT token should be zero

Severity: Informational

Context: PoolCreationHelper.sol#L119

Description: Per the Balancer SDK, the BPT balance in the initial balance is set to zero. Refer to Line 266 below.

- composable-stable.factory.ts#L266:

```
265:     const tokensWithBpt = [...tokensIn, poolAddress];
266:     const amountsWithBpt = [...amountsIn, '0'];
267:     const maxAmountsWithBpt = [
268:       ...amountsIn,
269:       // this max amount needs to be >= PREMINT - bptAmountOut,
270:       // The vault returns BAL#506 if it's not,
271:       // PREMINT is around 2^111, but here we set the max amount of BPT as MAX_UINT_256-1
↪  for safety
272:       BigInt.asUintN(256, BigInt(-1)).toString(),
273:     ];
274:     const [sortedTokens, sortedAmounts, sortedMaxAmounts] =
275:       assetHelpers.sortTokens(
276:         tokensWithBpt,
277:         amountsWithBpt,
278:         maxAmountsWithBpt
279:       ) as [string[], string[], string[]];
280:
281:     const userData = ComposableStablePoolEncoder.joinInit(sortedAmounts);
```

The `amountsWithBpt` array is sorted into `sortedAmounts` array, and then passed into the `ComposableStablePoolEncoder.joinInit` function to construct the `userData`. This `userData` will be used when initializing the pool.

- encoder.ts#L29:

```
29:   static joinInit = (amountsIn: BigNumberish[]): string =>
30:     defaultAbiCoder.encode(
31:       ['uint256', 'uint256[]'],
32:       [ComposableStablePoolJoinKind.INIT, amountsIn]
33:     );
```

However, within the `PoolCreationHelper` contract, the BPT balance in the initial balance is set to `2**(111)` instead.

- PoolCreationHelper.sol:

```
114:         // Copy existing initialBalances and insert preminted BPT at bptIndex
115:         for (uint256 i = 0; i < updatedInitialBalances.length; i++) {
116:             if (i < bptIndex) {
117:                 updatedInitialBalances[i] = initialBalances[i];
118:             } else if (i == bptIndex) {
119:                 updatedInitialBalances[i] = 2**(111); // PREMINTED_BPT_TOKEN_BALANCE
120:             } else { // @audit-info if i > bptIndex
121:                 updatedInitialBalances[i] = initialBalances[i - 1];
122:             }
123:         }
```

It is not necessary to define the initial balance of BPT tokens during initialization. Per the balancer's source code below, the number of `bptAmountOut` BPT tokens minted to the caller will be computed based on StableMath's invariants, as per Line 588 below. The remaining `_PREMINTED_TOKEN_BALANCE - bptAmountOut` BPT tokens will be minted to the caller and then pulled back to the vault, as per Lines 600-603 below.

Thus, the codebase automatically computes the BPT balance:

- ComposableStablePool.sol:

```
571:     function _onInitializePool(
572:         bytes32,
573:         address sender,
574:         address,
575:         uint256[] memory scalingFactors,
576:         bytes memory userData
577:     ) internal override returns (uint256, uint256[] memory) {
578:         StablePoolUserData.JoinKind kind = userData.joinKind();
579:         _require(kind == StablePoolUserData.JoinKind.INIT, Errors.UNINITIALIZED);
580:
581:         // AmountsIn usually does not include the BPT token; initialization is the one time
↪  it has to.
582:         uint256[] memory amountsInIncludingBpt = userData.initialAmountsIn();
583:         InputHelpers.ensureInputLengthMatch(amountsInIncludingBpt.length,
↪  scalingFactors.length);
584:         _upscaleArray(amountsInIncludingBpt, scalingFactors);
585:
586:         (uint256 amp, ) = _getAmplificationParameter();
587:         uint256[] memory amountsIn = _dropBptItem(amountsInIncludingBpt);
588:         uint256 invariantAfterJoin = StableMath._calculateInvariant(amp, amountsIn);
589:
590:         // Set the initial BPT to the value of the invariant
591:         uint256 bptAmountOut = invariantAfterJoin;
592:
593:         // BasePool will mint bptAmountOut for the sender: we then also mint the remaining
↪  BPT to make up the total
594:         // supply, and have the Vault pull those tokens from the sender as part of the
↪  join.
595:         // We are only minting half of the maximum value - already an amount many orders of
↪  magnitude greater than any
596:         // conceivable real liquidity - to allow for minting new BPT as a result of regular
↪  joins.
597:         //
598:         // Note that the sender need not approve BPT for the Vault as the Vault already has
↪  infinite BPT allowance for
599:         // all accounts.
600:         uint256 initialBpt = _PREMINTED_TOKEN_BALANCE.sub(bptAmountOut);
601:
602:         _mintPoolTokens(sender, initialBpt);
603:         amountsInIncludingBpt[getBptIndex()] = initialBpt;
604:
```

```
605:         // Initialization is still a join, so we need to do post-join work.
606:         _updatePostJoinExit(amp, invariantAfterJoin);
607:
608:         return (bptAmountOut, amountsInIncludingBpt);
609:     }
```

Recommendation: Consider updating the initialization process to be aligned with the Balancer's SDK implementation.

The `updatedInitialBalance[BPT_INDEX]` can either be set to zero (see composable-stable.factory.ts#L266) or left uninitialized. However, the BPT field of the maxAmountsIn should be set to PREMINTED_BPT_TOKEN_BALANCE.

Berachain: Fixed in PR 4.

Spearbit: Fixed. `amountsWithBpt[bptIndex]` has been initialized to zero, and `maxAmountsWithBpt[bptIndex]` has been set to `_MAX_UINT256` (`type(uint256).max`).

5.2.5 `PoolCreationHelper` does not allow for joining a WETH pool with ETH

Severity: Informational

Context: PoolCreationHelper.sol#L50

Description: The vault system in Balancer allows users to join pools, including WETH, by transferring WETH or native ETH. This is documented in the `IVault` interface.

```
/*
* If joining a Pool that holds WETH, it is possible to send ETH directly: the Vault will do the
↪  wrapping. To enable
* this mechanism, the IAsset sentinel value (the zero address) must be passed in the `assets` array
↪  instead of the
* WETH address. Note that it is not possible to combine ETH and WETH in the same join. Any excess ETH
↪  will be sent
* back to the caller (not the sender, which is important for relayers).
*/
```

However, the current implementation of `PoolCreationHelper` does not support this as neither `createAndJoinWeightedPool()` nor `createAndJoinStablePool()` are payable or implement forwarding of `msg.value`.

Recommendation: If this functionality is also intended to be supported, additional functionality would need to be implemented to support native transfers. Alternatively, it should be documented that this kind of joining pool is not supported.

Berachain: Documentation added in PR 6.

Spearbit: Fixed by the PR provided by the protocol.

5.2.6 `PREMINTED_BPT_TOKEN_BALANCE` should be defined as a constant

Severity: Informational

Context: PoolCreationHelper.sol#L119

Description: When users create and join a stable pool using `createAndJoinStablePool()`, we set the `updatedInitialBalances[bptIndex]` to the hardcoded value `2**(111)`.

```
// Copy existing initialBalances and insert preminted BPT at bptIndex
for (uint256 i = 0; i < updatedInitialBalances.length; i++) {
    if (i < bptIndex) {
        updatedInitialBalances[i] = initialBalances[i];
    } else if (i == bptIndex) {
        updatedInitialBalances[i] = 2**(111); // PREMINTED_BPT_TOKEN_BALANCE //<=  HERE
    } else {
        updatedInitialBalances[i] = initialBalances[i - 1];
    }
}
```

Hardcoding values are not recommended, as it leads to harder maintainable code. This can also lead to issues if the value changes in the future, but changing it here is forgotten because it is hidden in the code.

Recommendation: We recommend defining the hardcoded value as a constant to allow for a cleaner codebase and an easier way to change it.

Berachain: Fixed in PR 4.

Spearbit: Mitigated by the fix provided by the protocol.