Zellic

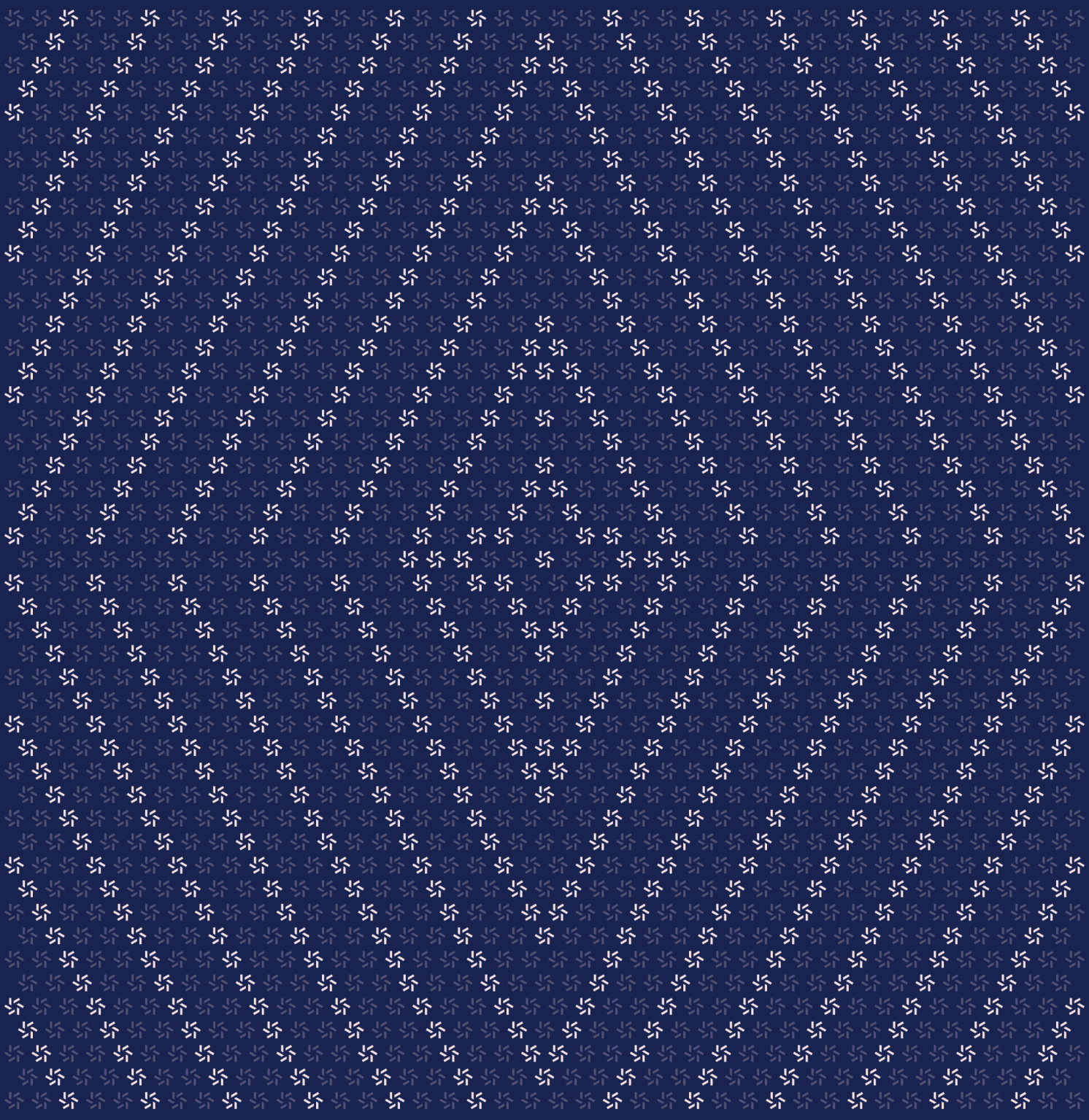
Prepared for
Bearlusconi Bera
Berachain

Prepared by
Katerina Belotskaia
Qingying Jie
Zellic

February 6, 2026

Berachain

Smart Contract Patch Review



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Berachain 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. The contract GuaranteedEmissionManager is missing a call to the function `_disableInitializers` in the constructor 11

3.2. Lack of duplicate receiver checks `_validateRewardAllocation` 12

4. Patch Review **13**

4.1. Notable changes 14

4.2. Minor differences 16

5. Threat Model **16**

5.1. Module: GuaranteedEmissionManager.sol 17

6. Assessment Results **22**

6.1. Disclaimer 23

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security patch review for Berachain from February 4th to February 5th, 2026. During this engagement, Zellic reviewed Berachain's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following question:

- Does the emission token (BGT) correctly distribute each time the `distributeFor` function is called (every block)?

We were asked to review a pull request (PR #90) that introduces a Guaranteed Emission (GE) stream for a foundation-managed subset of reward vaults. The GE stream is distributed in parallel to the existing proof-of-liquidity (POL) reward flow, enabling controlled and predictable emissions for selected vaults. The purpose of this review was to focus exclusively on changes introduced by this pull request to the Berachain contracts. In section 4. ↗, we have provided an overview of the changes.

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Contracts from the src/old_versions/V0_Contracts directory
- Front-end components
- Infrastructure relating to the project
- Key custody

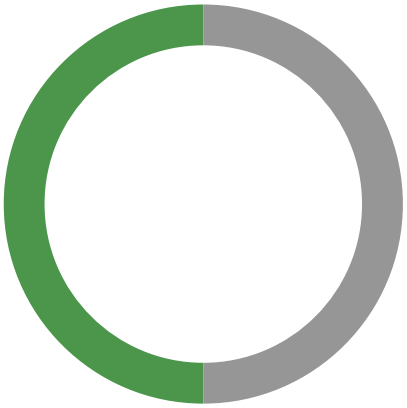
Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Berachain contracts, we discovered two findings. No critical issues were found. One finding was of low impact and the other finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	1



2. Introduction

2.1. About Berachain

Berachain contributed the following description of Berachain:

> Berachain is an EVM-identical Layer 1 (L1) blockchain that uses liquidity to secure the network and fuel the application layer. Unlike most blockchains that rely solely on traditional staking for network security, Berachain's novel Proof of Liquidity (PoL) consensus mechanism rewards users who actively contribute resources with the network's native governance token, $BGT.
>
> Berachain offers an EVM-identical execution environment, and leverages the novel BeaconKit on the consensus layer to provide single-slot finality. BeaconKit was developed by the Berachain Foundation to provide a modular consensus layer that is adaptable for teams wishing to build Ethereum-based blockchains.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability

weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Berachain Contracts

Type	Solidity
Platform	EVM-compatible
Target	Only the changes made in PR #90 up to the commit below.
Repository	https://github.com/berachain/contracts-internal/pulls/90 ↗
Version	`d0ecd47369de509e94bb7e1f3210ad069c9b2fde`

Programs	`GuaranteedEmissionManagerDeployer.sol`
	`BeraChef.sol`
	`Distributor.sol`
	`GuaranteedEmissionManager.sol`
	`RewardAllocatorFactory.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of three person-days. The assessment was conducted by two consultants over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Qingying Jie
Engineer
qingying@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 4, 2026	Start of primary review period
February 5, 2026	End of primary review period

3. Detailed Findings

3.1. The contract GuaranteedEmissionManager is missing a call to the function `_disableInitializers` in the constructor

Target	GuaranteedEmissionManager		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The contract GuaranteedEmissionManager is a upgradable contract, but it does not call the function `_disableInitializers` in its constructor.

```
contract GuaranteedEmissionManager is IGuaranteedEmissionManager,
    AccessControlUpgradeable, UUPSUpgradeable
```

Impact

Because the implementation contract does not call the function `_disableInitializers` in the constructor, its key state variables can be initialized to arbitrary values by anyone, which also violates best practices.

Recommendations

Consider adding the constructor with a call to the function `_disableInitializers`.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit 8d492329 ↗.

3.2. Lack of duplicate receiver checks `_validateRewardAllocation`

Target	GuaranteedEmissionManager		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `_validateRewardAllocation` function takes a `rewardAllocation` array of `Weight` structures as input. The `Weight` structure contains the `receiver` (vault) address and `percentageNumerator` — the fraction of rewards going to this receiver.

```
struct Weight {
    // The address of the receiver that this weight is for.
    address receiver;
    // The fraction of rewards going to this receiver.
    // the percentage denominator is: ONE_HUNDRED_PERCENT = 10000
    // the actual fraction is: percentageNumerator / ONE_HUNDRED_PERCENT
    // e.g. percentageNumerator for 50% is 5000, because 5000 / 10000 = 0.5
    uint96 percentageNumerator;
}
```

This function ensures that the total weight (the sum of the `percentageNumerator`) does not exceed 100%. However, it does not validate that all receivers are unique.

After successful validation, the global `_rewardAllocation` array will be updated by the new values and will be used during reward distribution.

Impact

During the reward distribution process, the amount of rewards a vault can receive depends on the `percentageNumerator` and the target emission set in the contract GuaranteedEmissionManager. First, the reward amount for each vault is calculated based on the `percentageNumerator` and the total rewards, and then the smaller value between this amount and the target emission is taken as the actual reward.

If the contract manager mainly relies on the `percentageNumerator` to allocate rewards to vaults and sets the target emission to a relatively large value, a vault with duplicate entries in the `_rewardAllocation` array may receive more rewards than expected in a single distribution.

Recommendations

Consider adding the validation check that `rewardAllocation` contains only unique vault addresses.

Remediation

This issue has been acknowledged by Berachain.

Berachain provided the following response to this finding:

> A more flexible validation approach was intentionally adopted compared to the standard reward allocation validation (e.g., no duplicate checks and no weight limits). This is an explicit design decision, as this reward allocation is managed internally and is not expected to be updated frequently. All updates are subject to a dedicated off-chain validation process. Additionally, a monitoring dashboard is being implemented to continuously track cash flows and detect any anomalous behavior. The on-chain whitelisted status check is intentionally preserved to ensure that all provided addresses correspond to vaults deployed via the factory and comply with the expected interface. Omitting this check could cause the distributeFor function to revert, which is an invalid and unacceptable execution path.

4. Patch Review

This section documents notable and minor changes applied to the in-scope code from commit a902a159 to commit d0ecd473.

4.1. Notable changes

The following were notable changes made to the codebase.

In the GuaranteedEmissionManager contract

The new GuaranteedEmissionManager contract has been added.

This contract allows to manage the emission percentage and reward allocation. Specifically, the `ALLOCATION_MANAGER_ROLE` is allowed to set up the `_rewardAllocation` using the `setRewardAllocation` function. The new weights will be validated to not exceed 100% in total.

After successful validation, the old `_rewardAllocation` data will be updated by the new array of `Weight` structures. It should be noted that the `Weight` structure, in addition to the fraction of rewards, also contains the receivers' addresses (can also be referred to as vaults' addresses). Every `receiver` address is verified using the method `beraChef.isWhitelistedVault()`, thereby allowing only trusted vaults to be specified.

```
function setRewardAllocation(Weight[] memory newRewardAllocation)
    external onlyRole(ALLOCATION_MANAGER_ROLE) {
    _validateRewardAllocation(newRewardAllocation);
    delete _rewardAllocation;
    uint256 length = newRewardAllocation.length;
    for (uint256 i; i < length;) {
        _rewardAllocation.push(newRewardAllocation[i]);
        unchecked {
            ++i;
        }
    }

    emit RewardAllocationSet(_rewardAllocation);
}
```

Additionally, the `ALLOCATION_MANAGER_ROLE` can set up the maximum emission for the vault. Although this function does not validate whether the vault is whitelisted, only trusted vaults can be set up to the `_rewardAllocation` and receive emission.

```
function setTargetEmission(address vault, uint256 _targetEmission)
    external onlyRole(ALLOCATION_MANAGER_ROLE) {
    if (debt[vault] > _targetEmission) {
        InvalidTargetEmission.selector.revertWith();
```

```
    }
    targetEmission[vault] = _targetEmission;
    emit TargetEmissionSet(vault, _targetEmission);
}
```

The `setEmissionPerc` function allows to specify the percentage of the total emission that will be allocated to these selected vaults.

```
function setEmissionPerc(uint96 _emissionPerc)
    external onlyRole(ALLOCATION_MANAGER_ROLE) {
    if (_emissionPerc > ONE_HUNDRED_PERCENT) {
        InvalidEmissionPerc.selector.revertWith();
    }
    emissionPerc = _emissionPerc;
    emit EmissionPercSet(emissionPerc);
}
```

During the reward distribution process, the distributor contract notifies every vault of the amount of rewards using the `notifyEmission` function, which keeps track of the remaining emission.

```
function notifyEmission(address vault, uint256 amount)
    external onlyDistributor {
    debt[vault] += amount;
    emit NotifyEmission(vault, amount);
}
```

See a more detailed description of the functions in section 5. ↗.

In the Distributor contract

The `setGuaranteedEmissionManager` function has been added, which allows the `DEFAULT_ADMIN_ROLE` to update the trusted GuaranteedEmissionManager contract address.

```
function setGuaranteedEmissionManager(address _guaranteedEmissionManager)
    external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (_guaranteedEmissionManager == address(0)) {
        ZeroAddress.selector.revertWith();
    }
    emit GuaranteedEmissionManagerSet(address(guaranteedEmissionManager),
    _guaranteedEmissionManager);
    guaranteedEmissionManager
    = IGuaranteedEmissionManager(_guaranteedEmissionManager);
}
```

Additionally, the `distributeFor` function has been updated, introducing a reward distribution process for the vaults provided by the guaranteed emission manager. The percentage is determined by the `guaranteedEmissionManager.emissionPerc`, and the remaining emissions will continue to be distributed to validators.

4.2. Minor differences

The following were minor changes made to the codebase.

In the GuaranteedEmissionManagerDeployer contract

The new GuaranteedEmissionManagerDeployer contract allows to deploy the GuaranteedEmissionManager using the proxy and initialize the initial state with the `owner`, `distributor`, and `beraChef` addresses.

```
function _deployGuaranteedEmissionManager(
    address owner,
    address distributor,
    address beraChef,
    Salt memory guaranteedEmissionManagerSalt
)
    internal
    returns (address)
{
    address guaranteedEmissionManagerImpl = deployWithCreate2(
        guaranteedEmissionManagerSalt.implementation,
    type(GuaranteedEmissionManager).creationCode
    );
    guaranteedEmissionManager = GuaranteedEmissionManager(
        deployProxyWithCreate2(guaranteedEmissionManagerImpl,
    guaranteedEmissionManagerSalt.proxy)
    );
    guaranteedEmissionManager.initialize(owner, distributor, beraChef);
    return address(guaranteedEmissionManager);
}
```

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: GuaranteedEmissionManager.sol

Function: `notifyEmission(address vault, uint256 amount)`

This function allows the `distributor` to notify a vault of an emission, which increases the `debt` of the vault by the specified amount.

Inputs

- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The vault address to notify.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The amount of emission to notify.

Branches and code coverage

Intended branches

- The `debt[vault]` is increased by `amount`.
  - ☑ Test coverage
- A `NotifyEmission` event is emitted with the `vault` and `amount`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the `distributor`.
  - ☑ Negative test

Function: `setBeraChef(address _beraChef)`

This function allows an account with the `DEFAULT_ADMIN_ROLE` to set the `beraChef` address.

Inputs

- `_beraChef`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The new `beraChef` address.

Branches and code coverage

Intended branches

- The state variable `beraChef` is updated to the new address.
  - ☑ Test coverage
- A `BeraChefSet` event is emitted upon successful execution.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have the `DEFAULT_ADMIN_ROLE`.
  - ☑ Negative test
- Reverts if `_beraChef` is the zero address.
  - ☑ Negative test

Function: `setDistributor(address _distributor)`

This function allows an account with the `DEFAULT_ADMIN_ROLE` to set the `distributor` address.

Inputs

- `_distributor`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The new distributor address.

Branches and code coverage

Intended branches

- The state variable `distributor` is updated to the new address.
  - ☑ Test coverage
- A `DistributorSet` event is emitted upon successful execution.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller does not have the `DEFAULT_ADMIN_ROLE`.
  - ☑ Negative test
- Reverts if `_distributor` is the zero address.
  - ☑ Negative test

Function: `setEmissionPerc(uint96 _emissionPerc)`

This function allows an account with the `ALLOCATION_MANAGER_ROLE` to set the emission percentage.

Inputs

- `_emissionPerc`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be greater than `ONE_HUNDRED_PERCENT` (1e4).
  - **Impact**: The new emission percentage.

Branches and code coverage

Intended branches

- The state variable `emissionPerc` is updated to the new value.
  - ☑ Test coverage
- An `EmissionPercSet` event is emitted upon successful execution.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller does not have the `ALLOCATION_MANAGER_ROLE`.
  - ☑ Negative test

- Reverts if `_emissionPerc` is greater than `ONE_HUNDRED_PERCENT`.
  - ☑ Negative test

Function: `setRewardAllocation(Weight[] newRewardAllocation)`

This function allows an account with the `ALLOCATION_MANAGER_ROLE` to set the reward allocation.

Inputs

- `newRewardAllocation`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: In the array `newRewardAllocation`, each `Weight` item has a `receiver` field and a `percentageNumerator` field. The `receiver` must be a whitelisted vault in the BeraChef contract, and the `percentageNumerator` must be greater than zero and less than or equal to `ONE_HUNDRED_PERCENT` (1e4). Additionally, the sum of all `percentageNumerator` values in the array must equal `ONE_HUNDRED_PERCENT`.
  - **Impact**: The new reward allocation.

Branches and code coverage

Intended branches

- The array `_rewardAllocation` is updated to the new value.
  - ☑ Test coverage
- A `RewardAllocationSet` event is emitted upon successful execution.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller does not have the `ALLOCATION_MANAGER_ROLE`.
  - ☑ Negative test
- Reverts if any `rewardAllocationItem.percentageNumerator` is zero or greater than `ONE_HUNDRED_PERCENT`.
  - ☐ Negative test
- Reverts if any `rewardAllocationItem.receiver` is not a whitelisted vault in the BeraChef contract.
  - ☑ Negative test
- Reverts if the total weight (sum of all `percentageNumerator` values) does not equal `ONE_HUNDRED_PERCENT`.

☑ Negative test

Function call analysis

- `this._validateRewardAllocation(newRewardAllocation) -> IBeraChef(this.beraChef).isWhitelistedVault(rewardAllocationItem.receiver)`
  - **What is controllable?** `rewardAllocationItem.receiver`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The transaction will be reverted if the `rewardAllocationItem.receiver` is not a whitelisted vault in the BeraChef contract.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setTargetEmission(address vault, uint256 _targetEmission)`

This function allows an account with the `ALLOCATION_MANAGER_ROLE` to set the target emission for a vault.

Inputs

- `vault`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The target vault address receiving the emission.
- `_targetEmission`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be less than the current `debt[vault]`.
  - **Impact**: The target emission amount for the vault.

Branches and code coverage

Intended branches

- The state variable `targetEmission[vault]` is updated to the new value.

  ☑ Test coverage
- A `TargetEmissionSet` event is emitted upon successful execution.

  ☑ Test coverage

Negative behavior

- Reverts if the caller does not have the `ALLOCATION_MANAGER_ROLE`.
  - ☑ Negative test
- Reverts if `_targetEmission` is less than the current `debt[vault]`.
  - ☑ Negative test

6. Assessment Results

During our assessment on the scoped Berachain contracts, we discovered two findings. No critical issues were found. One finding was of low impact and the other finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.