Retro/Thena Audit

OPENZEPPELIN SECURITY | JUNE 26, 2023

Security Audits

June 26, 2023

This security assessment was prepared by **OpenZeppelin**.

Table of Contents

Summary

Type
: DeFi

Timeline
: From 2023-05-01
: To 2023-06-06

Languages
: Solidity

Total Issues
: 47 (32 resolved, 8 partially resolved)

Critical Severity Issues
: 1 (1 resolved)

High Severity Issues
: 4 (3 resolved, 1 partially resolved)

Medium Severity Issues
: 9 (6 resolved, 1 partially resolved)

Low Severity Issues
: 12 (7 resolved, 3 partially resolved)

Notes & Additional Information
: 19 (13 resolved, 3 partially resolved)

Scope

We audited the ThenafiBNB/Thena-Contracts repository at the `2bb84cf6bb429e9bac8ac8464f4546da855ef4ca` commit.

In scope were the following contracts:

```
├── GaugeV2.sol
├── GaugeV2_CL.sol
├── PermissionsRegistry.sol
└── VoterV3.sol
```

Audit Context

The scope of this audit relates to a collaborative work between two protocols from different chains: Retro (Polygon) and Thena (BNB Chain). Retro has created a "friendly fork" from Thena on the Polygon blockchain and both entities are commited to an ongoing partnership including a shared roadmap. The dual audit was conducted to validate the reliability of this new, shared codebase and to assess and improve its security properties allowing both teams to collaboratively develop and advance their respective platforms.

System Overview

Thena is a DEX inspired by Solidly. The motivation behind Thena is to create a decentralized exchange (DEX) with enhanced capital efficiency for the liquidity providers compared to the traditional DEXs (Uniswap, Curve) with the added ability for external DeFi protocols to incentivize users to provide liquidity via rewards.

Thena supports three kinds of AMM pools: stable (similar to CurveV1), volatile (similar to UniswapV2), and concentrated liquidity (powered by Algebra). The liquidity providers do not immediately receive part of the pool's swap fees. Instead, they are incentivized to deposit their LP tokens to a `Gauge` contract in order to farm `$THE` emissions. Each `Gauge` corresponds to a specific pool. As long as they deposit their LP tokens in the `Gauge`, they are entitled to a portion of the `$THE` emissions distributed once per voting period. At the end of each voting period, the `$THE` amount that each gauge receives depends on its share of votes received. `veTHE` is the voting token and can be minted by locking `$THE` tokens.

In essence, liquidity providers are incentivized to deposit their LP tokens to the corresponding gauge, receive `$THE` emissions, lock their `$THE` tokens to receive `veTHE` and vote for their gauge so that it is entitled to a larger portion of the upcoming `$THE` emissions and so on.

pool they voted for. As a result, liquidity providers are incentivized both by the swap fees and the `$THE` emissions, assuming liquidity providers lock their `$THE` tokens into `veTHE` positions. Furthermore, external entities are able to provide extra rewards to either the voters or the LP token depositors in order to enhance the incentives for providing liquidity to a pool of interest.

Privileged Entities & Roles

There are several privileged entities and roles that have access to sensitive operations as follows.

The `thenaMultisig`:

- Can add new privileged roles.
- Can remove an existing privileged role.
- Can assign a role to a specific address.
- Can revoke a role from a specific address.
- Can change the address of `emergencyCouncil`.
- Can change the address for itself.

At the current version of the deployed `PersmissionsRegistry` contract on mainnet, `thenaMultisig` is a 4-out-of-6 multisig.

The `thenaTeamMultisig`:

- Is the receiver of rescued funds from the `CLFeesVault` contract in case of emergency.
- Is the owner of the `Bribes` contracts.
- Can add new reward tokens in the `Bribes` contract.
- Can recover any token's balance of the `Bribes` contract and transfer it to itself.
- Can recover any token's balance of the `Bribes` contract, transfer it to itself and update the epoch.
- Can set a new `Voter` address in the `Bribes` contract.
- Can set a new `Minter` address in the `Bribes` contract.
- Can set a new owner address for the `Bribes` contract.
- Can change the address for itself.

The `owner` (which initially is the deployer) of the `BribesFactoryV3` contract:

- Can update the reward amount to be distributed within a distribution period.
- Can set the reward rate to any arbitrary value.
- Can move amounts of any ERC20 token out of the contract.
- Can pause and unpause the contract.

The `emergencyCouncil`:

- Can activate the emergency mode for the contracts `GaugeFactoryV2_CL`, `GaugeFactoryV2`
- Can deactivate the emergency mode for the contracts `GaugeFactoryV2_CL`, `GaugeFactoryV2`
- Can change the address for itself.

At the current version of the deployed `PersmissionsRegistry` contract on mainnet, both `thenaTeamMultisig` and `emergencyCouncil` are set to the same 2-out-of-2 multisig.

The `GOVERNANCE` role:

- Can whitelist an LP token for gauge creation.
- Can remove an LP token for gauge creation from the whitelist.
- Can pause a gauge.
- Can revive a paused gauge.
- Can completely remove a gauge.

The `VOTER_ADMIN` role:

- Can provide initialization values to important parameters of the `VoterV3` contract.
- Can configure `voteDelay`, `Minter` address, `BribeFactory` address, `GaugeFactoryAddress`, `PairFactory` address, `PersmissionsRegistry` address, set new `Bribes` contracts, add/replace/remove pair and gauge factories.
- Can force reset a `tokenId`'s `lastVoted` value.

The `GAUGE_ADMIN` role:

- Is allowed, for a gauge of a concentrated-liquidity pool, to set the `PermissionsRegistry` address, set the fee vault's address and set the address of the default fee recipient.

The `owner` (which is initially the deployer) of the `GaugeFactoryV2` contract:

- Has similar privileged access rights on the `Gauge` contracts as `GAUGE_ADMIN`.

The `BRIBE_ADMIN` role:

- Is allowed to add a reward token to a `Bribes` contract.

The `CL_FEES_VAULT_ADMIN` role:

- Can configure important parameters of the `CLFeesVault` contract as well as transfer the `CLFeesVault` contract's balance of any token to itself.

At the current version of the deployed `PersmissionsRegistry` contract on mainnet, all roles `GOVERNANCE`, `VOTER_ADMIN`, `GAUGE_ADMIN`, `BRIBE_ADMIN` and `CL_FEES_VAULT_ADMIN` are set to the same EOA account.

Security Model and Trust Assumptions

The contracts in scope rely on access control to manage and operate the smart contracts. As some of the administrative operations are highly sensitive, the parties who operate them are seen as trusted. However, we note that major risks may arise in the case that a privileged entity acts maliciously.

Entity `thenaMultisig`:

- Could assign any privileged role to a malicious address.
- Could set privileged entity `emergencyCouncil` to a malicious address.

Entities `thenaTeamMultisig` and `owner` of the `BribeFactoryV3` contract:

- Could arbitrarily activate the emergency mode for the `Gauge` contracts, possibly resulting in complete DoS for these contracts.

Entity `owner` of the `GaugeExtraRewarder` contract:

- Could set an arbitrary reward rate value, possibly resulting in unfair reward distribution.
- Could use the ERC20token recovery function to drain the contract's reward funds.

Role `GOVERNANCE`:

- Could arbitrarily pause or completely remove a gauge causing loss of reward funds.

Role `VOTER_ADMIN`:

- Could arbitrarily configure the value of a user's `lastVoted` timestamp affecting their vote accounting.
- Could set arbitrary addresses for any of the contract `Minter`, `BribeFactory`, `GaugeFactory`, `PairFactory` `PermissionsRegistry` in the `VoterV3` contract. This could have serious effects like complete DoS and draining all of the contract's funds.

Role `GAUGE_ADMIN` and entity `owner` of the `GaugeFactoryV2` contract:

- Could maliciously change the contract addresses of `GaugeExtraRewarder`, `RewardDistribution` and `Bribes` for a gauge, potentially resulting in draining all of the reward amounts in a gauge.

Role `CL_FEES_VAULT_ADMIN`:

- Could transfer all accrued fees in the `CLFeesVault` contract to `thenaTeamMultisig`.

Given that at the current status of the deployed contracts, `thenaMultisig` is a 4-out-of-6 multisig, `thenaTeamMultisig` and `emergencyCouncil` are the same 2-out-of-2 multisig and all of the privileges roles are resolved to the same EOA account, we note that the role distinction is in practice unclear and decentralization should be enhanced.

Vulnerabilities targeting this part of the system were not considered in the list of issues.

Update: *The Retro-Thena team stated:*

> *We agree on the necessity of increasing decentralization. We're going to integrate governance and increase decentralization in the long run. Neither our team nor any multisig can touch users' LP tokens. All the addresses set to the* `PermissionsRegistry` *contract are used for daily operations (eg.: whitelisting bribes, tokens, adding extra rewarders,..)*
>
> *Currently: - Thena Team Multisig: bnb address:0x46F99291Eedf25fd5c6AE56BbfD6679d0eA3630B is a 2/2 signature, composed by Prometheus (Lead dev) and Theseus (co-founder) - Thena Multisig: bnb address:0x7d70ee3774325C51e021Af1f7987C214d2CAA184 is a 4/6 signature, composed by Prometheus, Theseus, Lafa (DEUS), Pratas.eth (DefiBD), Andrés (Grizzly.fi), H.P. (FE/BE Thena dev) - 0x993Ae2b514677c7AC52bAeCd8871d2b362A9D693 EOA: This is Thena's Deployer address. It's a cold wallet managed by Prometheus.*

Project Design and Maturity

The main functionalities of the protocol share the same design as its parent project, Velodrome. The current version of the codebase shows room for improvement for the following main reasons:

- There are several instances of code duplication across contracts and functions.
- There is insufficient test coverage, with an overall mean value of less than 50% of line coverage and less than 20% of branch coverage.
- There are instances where the codebase could be simplified or written with more clarity.
- There is a poor distinction between privileged roles which impose serious trust assumptions.

In addition, the issues that were discovered during this audit (1 Critical, several High and a number of Medium) suggest that changes may be needed in some important parts of the protocol. In order to reach a satisfying level of confidence for the project's overall security, it is necessary to expand the test suite and significantly improve the code coverage percentage.

Gauges Rewards Can Be Erroneously Deleted

Each voting period is supposed to last 1 week. There are two ways for anyone to update the system's period after this time has passed:

1. Call one of the `distribute` functions of `VoterV3`, which in turn calls the `Minter` contract's `update_period` function
2. Call `update_period` on the `Minter` contract directly

The second method to update the system's period can allow attackers to delete entire reward distributions which were allocated to gauges. This is attributed to a bug in the `_updateFor` function, where the calculation incorrectly considers the timestamp of the current active epoch when counting votes, rather than the previous epoch.

More specifically, consider the following attack scenario:

First, the function `update_period` is called directly on the `Minter` contract and executes successfully. This means that the active period changes, the gauges' reward for the finished period is notified to the Voter contract, and the `index` is increased.

Later, the `vote` (or `poke`) function is called. As a result, the function `_updateFor` is triggered for each voted gauge. During this update calculation, the total votes for `_epochTimestamp` are equal to zero, as this is the very first vote for the new period. In consequence, `supplyIndex` for the gauge is set equal to `index` without accounting the claimable amount for the period that just ended, making it impossible for the gauge to ever claim that amount.

An attacker could exploit this vulnerability to cause a complete DoS on the gauges' rewards by voting a small amount on every gauge before their `distribute` functions are called.

Consider using the previous period's timestamp for calculations in the `_updateFor` function. As an alternative solution, consider completely removing the gauge's update functionality upon every voting action and relying on the updates triggered by the Voter's `distribute` functions, which are supposed to take place once per voting period.

High Severity

`GaugeExtraRewarder` Accounts for Excessive Rewards

The function `updatePool` of the `GaugeExtraRewarder` contract is supposed to update the pool's information upon every state-changing interaction with the contract. The reward-per-share value is then updated by considering the time interval since the latest update. However, the reward rate value keeps increasing even if the distribution period is over. In this scenario, the variable `lastRewardTime` is assigned incorrectly, as it should be set equal to the end of the distribution period.

This improper accounting results in users accruing and taking rewards that were meant for other users, and in the worst case, causing the underlying gauge to revert during deposit and withdrawal operations.

Consider updating the implementation of the `GaugeExtraRewarder` contract so as to properly account for the reward amounts after the end of the distribution period.

Update: *Resolved in pull request #3 at commit 6d740eb.*

Killing a Gauge Could Result in Stuck Funds

When a gauge is killed using the `killGauge` or `killGaugeTotally` functions in the `VoterV3` contract, the gauge-related data is cleared out. However, it is possible that the killed gauge has already been voted for and its respective vote weight has been added to the period's total vote weight. The period's total weight is not adjusted when a gauge is killed, resulting in the portion of the reward that is committed to that gauge remaining unused and getting stuck in the contract.

More specifically, there are two possible problematic scenarios, regarding the rewards accounting of the current and previous period relative to the time that a gauge is killed.

1. **Effects on the currently active voting period.** If a gauge is killed in the middle of a period, it's possible that this gauge has already been voted for by some users, so it already contributes to `totalWeightPerEpoch`, which is later used to compute the `index` for that period. As a consequence, the killed gauge's share will eventually remain stuck in the

2. **Effects on the previous voting period.** It is possible that a gauge is killed at a time when a voting period has been updated but the reward amounts have not been distributed yet. This is possible because updating the period and distributing the respective rewards to the gauges require two separate actions (period update and rewards distribution). In this scenario, the `totalWeightsPerEpoch` that has been used to compute `index` includes the killed gauge's weight, even if its share of rewards will not be attributed to it after it's killed. Thus its share will eventually remain unused and stuck in the `VoterV3` contract.

When a gauge is killed, consider updating the `totalWeightsPerEpoch` value of the currently active voting period in order to distribute the whole reward amount to the remaining gauges. In addition, consider consolidating the reward amount notification and the distribution of the gauges' rewards into one single action in order to avoid losing part of the previous period's rewards when a gauge is killed.

Update: *Partially resolved in pull request #3 at commit 7864885. The effects on the currently active voting period are handled by updating* `totalWeightPerEpoch` *when killing a gauge. Regarding the effects on the previous voting period, the Retro-Thena team stated:*

> *We do not expect any gauge to be killed between a period update and rewards distribution because of our off-chain procedures that make sure to trigger the rewards distribution right after each period update.*

Killed Gauge Can Be Voted For

The `VoterV3` contract has safety measures put in place, allowing for the temporary and permanent removal of gauges in the case they are malicious. In order to track the state of a gauge, the `isAlive` mapping will be set to either `true` or `false` depending on whether a gauge is alive or killed respectively. Both the temporary and permanent removal of gauges sets the `isAlive` mapping to `false`.

During the voting phase, there only exists a check to see if the address is a gauge, which doesn't take into account if the gauge is still alive. This check will pass for any gauge temporarily killed, as a temporarily killed gauge will still reside in the `isGauge` mapping. This means users can vote for killed gauges.

gauges will lock funds on the [illegible] contract.

Consider removing the ability to vote from a killed gauge in order to avoid locking funds on the `VoterV3` contract.

Update: *Resolved in pull request #3 at commit 64bf6c5.*

Users Can Be Locked From Voting

In the `VoterV3` contract, users vote in order to receive rewards from `gauges`. During each epoch, rewards are reset and users need to vote again, either by calling the `poke` or `vote` functions. If a user does not vote in an epoch, they will not receive rewards. Votes do not roll over from epoch to epoch.

Through each method of voting, the `_reset` function is first called, clearing out the `tokenId`'s previous vote allocation to the gauges. During this reset, internal and external bribes from the votes are cleared by calling the `withdraw` function of the `Bribes` contract.

However, if a gauge is destroyed via the `killGaugeTotally` function, these internal and external `Bribes` addresses will be cleared while leaving the `poolVote` mapping populated with votes for pools that are linked to non-existent gauges. This results in the `withdraw` calls reverting for any voter escrow token that contains uncleared votes for a gauge killed with `killGaugeTotally`.

Interestingly, these calls should fail during the `gauges[_pool]` lookup in the `_withdraw` execution, as that is supposed to be cleared during `killGaugeTotally` call, but due to another bug, this will return the killed gauge's address.

Similarly, this should be able to be remedied by creating a new gauge for the pool, but in the `_createGauge` function, there is a requirement for the `gauges` mapping to hold the zero address, which will not be the case due to the previously mentioned bug.

In order to avoid locking users out of voting, consider adding logic to avoid reverts during `_reset` calls to tokens which have outstanding votes to totally killed gauges.

Medium Severity

`GaugeExtraRewarder`'s Reward Rate Can Be Set to an Arbitrary Value

The owner of the `GaugeExtraRewarder` contract is able to set an arbitrary value as reward rate. However, the reward rate should always be calculated based on the existing reward amount and the distribution period otherwise inconsistent accounting is possible, resulting in unfair rewards distribution among the users.

Consider removing the function `setRewardPerSecond` and always using `setDistributionRate` instead in order to avoid inconsistent rewards accounting.

Update: *Resolved in pull request #4 at commit e8b757f.*

Reward Amount in `GaugeExtraRewarder` Can Be Overestimated

The function `setDistributionRate` of the `GaugeExtraRewarder` contract first checks that the contract's balance is sufficient to cover the notified reward amount. However, any remaining reward amount from the latest distribution period is not considered during this check, as it is added to the total distributed reward amount only after this check. As a consequence, the final reward amount to be distributed may exceed the contract's reserves.

Consider ensuring that the total reward amount can be covered by the contract's reserves.

Update: *Resolved in pull request #4 at commit a1dc83f.*

Wrong Accounting of Extra Rewards Upon Depositing to a Gauge

Liquidity providers that deposit their tokens to a gauge are entitled to a portion of the extra rewards, if there are any. The extra reward amount is handled by the GaugeExtraRewarder contract and is distributed to the users in proportion to their share of the total staked LP tokens. Thus, each time the total staked amount in the gauge changes (i.e. upon deposit or withdrawal) the extra-reward-per-share value needs to be updated appropriately.

Upon depositing to a gauge, the user's funds are first transferred to the contract and the `onReward` function of the `GaugeExtraRewarder` contract is only called thereafter. The

reward.

Consider transferring the user's deposited funds only after calling the `onReward` function.

Update: *Resolved in pull request #4 at commit da9df65.*

Error-prone Contracts Initialization

The `VoterV3` contract has two separate initialization functions, `initialize` and `_init`. It is possible for users to interact with this contract in the invalid state where the `initialize` function has been called but the `_init` function has not.

In addition, several other issues appear:

- `initialize` sets `minter` and `permissionRegistry` equal to `msg.sender`. This is incorrect as neither of these contracts are supposed to initialize the Voter contract, nor should they be the same contract.
- `_init` is required to be called by the `minter` or the `permissionRegistry`, though neither of these contracts supports this functionality.

On a similar note, the constructor of the `PermissionsRegistry` contract sets all three entities `thenaTeamMultisig`, `thenaMultisig`, and `emergencyCouncil` equal to `msg.sender`. While this is technically feasible (e.g. all three lie behind the same multisig), it seems unreasonable to have all three roles handled by the same entity.

Consider having a single initialization function for all upgradeable contracts and assigning reasonable values to the system's parameters during initialization, so the protocol can immediately function in a secure manner.

Update: *Acknowledged, not resolved. The Retro-Thena team stated:*

> *Users cannot interact with the protocol before `_init` is called. This is because `createGauge` would fail since no token will have been whitelisted yet and `vote`/`reset`/`poke` would also fail since the `minter` contract, used in*

> *before actually going live.*

Risk-prone Accounting in `GaugeExtraRewarder`

The `GaugeExtraRewarder` contract tracks shares by updating internal tracking whenever the `onReward` function is called. This function is called from the underlying gauge whenever tokens are deposited or withdrawn.

However, if any deposits or withdrawals are executed while the `GaugeExtraRewarder` is paused, the `onReward` function will not track the state changes, resulting in the internal accounting of the `GaugeExtraRewarder` not matching the internal accounting of the gauge. Additionally, the internal accounting can become inaccurate if the gauge ever changes its `GaugeExtraRewarder`, or if withdrawals and deposits are done before setting a `GaugeExtraRewarder`.

Consider updating the accounting method of the `GaugeExtraRewarder` to accurately track shares on their associated gauge contracts. In order to save gas fees, it may be beneficial to piggyback off of the accounting done on the gauge rather than tracking everything in two places.

Update: *Partially resolved in pull request #4 at commit 6b17ff7. The pause functionality for the `GaugeExtraRewarder` contract has been removed. The accounting method has not been altered, which requires that the users interact with the system when a "GaugeExtraRewarder" contract with extra reward amount is deployed in order to start accruing their extra reward share.*

`gauges` Mapping Cannot Be Cleared

When calling `killGaugeTotally`, the `_gauge` key in the `poolForGauge` mapping is first deleted then later retrieved and used as a key to clear the `gauges` mapping. This retrieval will always return the zero address, resulting in the pool address never being cleared from the `gauges` mapping.

Additionally, this results in the inability to create a new gauge for a pool whose gauge was killed totally, as the mapping will already have an address set for the given pool.

Update: *Resolved in pull request #3 at commit 35c01c8.* `killGaugeTotally` *has been completely removed.*

Whitelist Is Incompatible With Proxies

The intention of the whitelist is to keep malicious contracts off of the protocol. When whitelisting upgradable contracts, it is possible for a formerly benign whitelisted contract to eventually upgrade into a malicious contract. Because of this, proxy contracts should never be whitelisted.

Consider removing any proxy contracts from the whitelist and introducing documentation around proxies in the whitelist code to avoid future proxies from being whitelisted.

Update: *Acknowledged, not resolved. The Retro-Thena team stated:*

> *We do due diligence on the whitelisted projects and we are in contact with them. Some projects (e.g., USDC) need a proxy.*

Removing a Role Does Not Remove the Role From All Users

The `removeRole` function in the `PermissionsRegistry` contract allows the Thena multisig to remove roles from the `PermissionsRegistry`.

When removing roles, the role is removed from the `_roles` list, but does not remove that role's entries in the `hasRole` mapping.

This is problematic because typical access control using the `PermissionsRegistry` contract looks up `msg.sender` in the `hasRole` mapping without first checking if the role is valid via either `_checkRole` or the `_roles` list. As a consequence, it is possible that formerly authorized entities can still perform sensitive operations.

Consider removing all users from the role upon deletion of the role.

Update: *Resolved in pull request #4 at commit 8b2b9c1.*

Unfair Distribution of Rewards

Reward distribution is calculated by directly increasing the `index` parameter. During an `_updateFor` call, which is called every time a vote is cast to a gauge, the rewards are allocated to the gauges when `claimable` is updated. Since `_updateFor` calculates the rewards based off of the current number of votes, the reward will be distributed in proportion to the snapshot of votes in the system when the funds were contributed.

This does not follow the rest of the protocol's calculations, where voters are given rewards at the end of the voting epoch and the votes' distribution is final. Additionally, this could lead to bizarre contributions where a user can contribute a large sum of rewards, but be the only voter in the epoch, resulting in all of the contributed funds being directed to them, signaling a large contribution but really not contributing anything.

In order to match other accounting done in the system and avoid the distribution of rewards based on intermediate voting state, consider changing the `_notifyRewardAmount` to distribute rewards at the end of the epoch rather than during one.

Update: *Resolved in pull request #4 at commit 8f108fd. `_notifyRewardAmount` has been completely removed.*

Low Severity

Invalid Entries Remain in `_addressToRoles` Array

The function `removeRoleFrom` of the `PermissionsRegistry` contract revokes a role from the specified `address` and is supposed to appropriately update the arrays `_roleToAddresses` and `_addressToRoles`.

However, because of a typographical error, the code never loops over the `_addressToRoles` array, resulting in invalid remaining entries.

Consider fixing the typographical error so that no invalid entries remain in the array.

Update: *Resolved in pull request #5 at commit 95d8d25.*

Unsigned Integer Variable Declared and Treated as a Signed Integer

monotonically increasing. This leads to unnecessary [illegible] and is also error prone in consideration of future updates to the codebase.

Consider declaring variable `rewardDebt` as an unsigned integer.

Update: *Resolved in pull request #5 at commit a2f45d0.*

Unsafe ABI Encoding

It is not an uncommon practice to use `abi.encodeWithSignature` or `abi.encodeWithSelector` to generate calldata for a low-level call. However, the first option is not typo-safe and the second option is not type-safe. The result is that both of these methods are error-prone and should be considered unsafe.

On line 861 of `VoterV3` an unsafe ABI encoding is used.

Consider replacing the unsafe ABI encoding with `abi.encodeCall`, which checks whether the supplied values actually match the types expected by the called function and also avoids errors caused by typos. Alternatively, consider changing the type of the `token` argument to an `IERC20` so normal function calling syntax can be used.

Update: *Resolved in pull request #5 at commit c3f6d95.*

Missing Docstrings

Throughout the codebase, there are several parts that do not have docstrings.

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Partially resolved in pull request #5 at commit 7ca7fc1.*

The System Is Not Robust to Delayed Period Updates

to a voting period is rather brittle.

More specifically, the unique identifier of a period is its starting timestamp, which is calculated modulo a week's time. When calculating the rewards of a voting period that has ended, that period's id is retrieved by manually subtracting a week's seconds from the current active period's id. However, it is not strictly imposed by the codebase that the time interval between two periods is exactly one week but rather a multiple of a week's time. It is possible, for example, that a period update action continuously fails for longer than a full period. In such a case, it would not be possible to access the past period's data and any reward distribution action would fail silently, essentially locking the reward amount in the contract.

Consider introducing a status for each voting period's phase, so as to be able to distinguish non-existing period timestamps due to delayed updates. As a more radical solution, consider adopting a serial number for uniquely identifying the voting periods and making the period's starting timestamp a separate data entry. This would make the system robust to potential delayed period updates but would also help improve the codebase's readability and overall quality making it less error-prone during future upgrades.

Update: *Acknowledged, not resolved. The Retro-Thena team stated:*

> *We ensure that each epoch lasts 1 week and no delays are allowed.*

Voting or Poking Emits Wrong Event

Users vote for gauges by calling the `vote` or `poke` function in the `VoterV3` contract. These functions emit the `Voted` event. These functions also call the internal function `_reset` which emits the `Abstained` event.

Hence, both `Voted` and `Abstained` events are emitted when a user votes. This could result in off-chain indexers computing the protocol state incorrectly.

Consider removing the `Abstained` event from the `_reset` function and only emitting it when a user abstains from voting.

Update: *Acknowledged, not resolved. The Retro-Thena team stated:*

Missing Error Messages in `require` Statements

Throughout the codebase, there are `require` statements that lack error messages. There are also some cases, for example line 244 in the `GaugeV2` contract, where the error messages are undermined by checks that overflow or underflow.

Consider including specific, informative error messages in `require` statements to improve overall code clarity and facilitate troubleshooting whenever a requirement is not satisfied. In addition, consider appropriately modifying the checks of the `require` statements wherever needed, so that it is not possible to overflow or underflow.

Update: *Partially resolved in pull request #5 at commit bd5dc9f and pull request #6 at commit a0eac56. There are still a few* `require` *statements that miss an error message.*

Code Simplifications

A number of opportunities for simplification of code were observed. Consider applying the following code simplifications:

- Contracts `GaugeV2` and `GaugeV2_CL` can be refactored so that a base contract contains the shared logic.
- Line 123 of the `GaugeExtraRewarder` contract can be simplified as the first condition is already covered by the second.
- The functions `balanceOfAt`, `balanceOf`, `balanceOfOwner`, and `balanceOfOwnerAt` can be refactored to consolidate shared logic.
- The functions `getReward` and `getReward` in the `GaugeV2` contract can be refactored to consolidate shared logic.
- The functions `earned`, `earned`, and `earnedWithTimestamp` in the `Bribes` contract can be refactored to consolidate shared logic.
- The functions `getReward`, `getReward`, `getRewardForOwner`, and `getRewardForAddress` in the `Bribes` contract can be refactored to consolidate shared logic.
- In the `_vote` function of the `VoterV3` contract, the `_usedWeight` variable contains the same value as `_totalWeight` is therefore redundant.

`VoterV3` contract can be refactored to consolidate shared logic.

- The function `claimBribes` on line 467 and `claimFees` on line 475 of the `VoterV3` contract are identical.
- The function `claimBribes` on line 483 and `claimFees` on line 490 of the `VoterV3` contract are identical.
- In line 366 and line 382 of the `_reset` function of the `VoterV3` contract, there is a call to `_epochTimestamp()` but the value of `_epochTimestamp()` has already been stored in the `_time` variable.
- The `_reset` function in the `VoterV3` contract can be refactored as follows:
  - The `if` statement on line 370 is redundant as `_votes` is unsigned and resides inside the check on line 362
  - The calls to `_withdraw` on lines 371-372 are only meaningful when the token was last voted in the current active period. If these operations are performed only when this condition is met, the `if` statement on line 382 can be removed.
- In the `GaugeV2` contract, the variables `fees0` and `fees1` are only ever assigned to zero and are therefore redundant.
- In the `Bribes` contract, the `if` statement on lines 131-133 is redundant, as this case is covered on lines 141-143.
- In the `Bribes` contract, the `IERC20Ext` interface declaration is unused and can be removed.
- In the `PermissionsRegistry` contract, the functions `__helper_stringToBytes` and `__helper_bytesToString` are unused and can be removed.
- In lines 821 and 844 of the `VoterV3` contract, the variable `_supplied` is cast to `uint` but is already declared as `uint256`.
- The `increaseGaugeApprovals` function of the `VoterV3` contract sets the allowance of `_gauge` by `VoterV3` to `type(uint256).max`. However, the allowance is already set to `type(uint256).max` during the gauge creation and the approval will never decrease. Therefore, the `increaseGaugeApprovals` function is redundant.

Lacking Address Checks

Across the repository, addresses are commonly saved into storage to be used later. When saving addresses, it is best practice to add basic checks to avoid accidentally setting these addresses to incorrect values.

In some cases, these addresses are expected to be set to the address of smart contracts. The best check for this case is using a code length check such as a library that provides an `isContract` function. Otherwise, a check to ensure the provided address is not the zero address is helpful.

Consider adding address checks to all addresses set in the codebase to help protect against the accidental setting of incorrect addresses.

Update: *Resolved in pull request #5 at commit 0f1551f.*

No Events Emitted on Sensitive Operations

There are plenty of functions throughout the codebase that are only called by privileged entities and perform sensitive operations, yet emit no events. Whenever executing highly privileged actions on-chain, it is preferable to emit some sort of logging for easy tracking and increased visibility. Some examples of such sensitive actions include setting a role for an address in the `PermissionsRegistry` contract, setting a new owner in the `Bribes` contract, and activating emergency mode in the `Gauge` contracts.

Consider emitting informative events upon each sensitive operation taking place in the system.

Update: *Resolved in pull request #5 at commit ebf6eaa and in pull request #10 at commit b369708.*

Recover Function Can Stall Associated Gauge

The `owner` of a `GaugeExtraRewarder` can use the `recoverERC20` function to move any ERC20 token from the contract. However, if an amount of the reward token is removed while there are active users, deposit and withdraw operations in the associated gauge contract can fail on the reward transfers due to low balance.

token needs to remain recoverable for emergency cases, consider adding a `require` statement to ensure that the `GaugeExtraRewarder` contract is stopped before recovering the reward token from the contract.

Update: *Resolved in pull request #9 at commit 748e153 and pull request #12 at commit 7693167. For the reward token, the owner can only recover up to the amount that has not yet been accounted for distribution. The Retro-Thena team stated:*

> *The `owner` of the `GaugeExtraRewarder` contract is either the project that provides the extra reward amount or the Thena-Retro team.*

Invalid Votes Are Counted Towards a User's Vote Weight

On line 428 of the `_vote` function in the `VoterV3` contract, the user's vote `_weights` are accumulated into the system's `_totalVoteWeight`, regardless of whether or not the vote is for a valid gauge. As a consequence, it is possible that only part of the user's voting power is utilized, especially for users who tend to repeat their voting across epochs using `poke`.

Consider reverting when a user attempts to vote for an address that is a totally killed, or otherwise invalid gauge.

Update: *Resolved in pull request #9 at commit 983da1f.*

Notes & Additional Information

Unused Function Arguments

The `onReward` function in the `GaugeExtraRewarder` contract has two unused arguments: `pid` and `extraData`.

To improve the overall clarity, intentionality, and readability of the codebase, consider removing the unused function parameters. This would simplify the calling contract's code as well.

Update: *Resolved in pull request #6 at commit 62a11ec.*

Hardhat `console` Import Present

Update: *Resolved in pull request #6 at commit 99574c5.*

Unused Imports

In the `Bribes` and `VoterV3` contracts there are some imports that are unused and could be removed:

- Import Math of contract `Bribes`
- Import Ownable of contract `Bribes`
- Import Math of contract `VoterV3`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #6 at commit 0fc5de4.*

Unused Event

In the `GaugeExtraRewarder` contract, the `LogOnReward` event is unused.

To improve the overall clarity, intentionality, and readability of the codebase, consider emitting or removing the unused event.

Update: *Resolved in pull request #6 at commit 158db13.*

Lack of Indexed Event Parameters

Throughout the codebase, several events do not have their parameters indexed. For instance:

- Line 439 of `Bribes.sol`
- Line 443 of `Bribes.sol`
- Line 73 of `VoterV3.sol`

Consider indexing event parameters to improve the ability of off-chain services to search for and filter for specific events.

Update: *Acknowledged, not resolved.*

The owners of LP tokens that deposit to a gauge are rewarded with `Thena` emissions. The reward amount is proportional to the amount of votes that the gauge receives. It is updated by the end of each voting period and is fully distributed within one week (i.e. until the next reward update). Each depositor is eligible for a portion of the reward in proportion to their share of the total deposits and the duration of their deposit with respect to the distribution period.

This design results in a misalignment of the depositors' contribution during a voting period and the corresponding received reward. For example, consider a user that has deposited during the whole voting period `i` and the reward amount emitted as a result of that period's votes, `r[i]`. The reward `r[i]` will be distributed during the voting period `i+1` among all the users that deposit during week `i+1`. The effects of this design are more obvious when considering the very first voting/deposit period of the system: the first reward amount, emitted at the end of the first period, will be distributed among the depositors of the second period. Therefore, the depositors of the first period receive no rewards. Furthermore, the portion of the reward received decreases as more depositors enter the gauge in the second period. More generally, the relation between each depositor's per-period contribution and the corresponding reward amount received for this contribution is not clear. This issue also holds for the extra rewards, if any, which are handled by the `GaugeExtraRewarder` contract.

Consider clearly documenting the misalignment between the depositors' per-period contribution and the corresponding rewards (and/or extra rewards) received. Also, consider documenting how the very first depositors in a gauge essentially receive no rewards by the end of the first period.

Update: *Acknowledged, not resolved. The Retro-Thena team stated:*

> *The gauges reward distribution is designed as a continuous streamline of rewards. We do not want to limit rewards to be distributed once per epoch as in the `Bribes` contracts. Before the creation of a new gauge, we provide an estimation of the expected rewards in the UI.*

Inconsistent and Outdated Solidity Versions

Different versions of Solidity are used across the contracts. In some contracts, the solidity version is outdated whereas in others the solidity version is floating.

aware that compiling and deploying on `ZKSync` may not work properly as this version introduces a new opcode - `PUSH0`.

Consider taking advantage of the latest Solidity version to improve the overall efficiency and security of the codebase. Regardless of the Solidity version used, consider keeping it consistent and locking it throughout the codebase to prevent the introduction of bugs due to incompatible future releases.

Update: *Resolved in pull request #6 at commit 406075d.*

Unnecessary Use of `SafeMath` Library

In some contracts, such as `GaugeV2`, `GaugeV2_CL`, and `GaugeExtraRewarder`, the protocol uses the OpenZeppelin SafeMath library for basic arithmetic functions, while using a compiler version above 0.8. However, `solc` versions above 0.8 contain built-in overflow and underflow protection.

To save gas and maintain consistency with the rest of the codebase, consider using Solidity's built-in arithmetic operators.

Update: *Resolved in pull request #6 at commit a0eac56 and in pull request #8 at commit ba97a1d.*

Unused State Variables

The `factory` and `gaugeFactory` variables in `VoterV3` contract are never used. As a result, their setter functions (`setPairFactory`, `setGaugeFactory`) are essentially dead code.

In addition, the `TYPE` variable and modifier `onlyOwner` in the `Bribes` contract are never used.

Similarly, `_VE` and `external_bribe` are unused in `GaugeV2` as well as `_VE` and `external_bribe` in `GaugeV2_CL`.

Consider removing the unused variables and their corresponding setter functions. If the variables are kept to maintain storage layout consistency with previous versions of the contract, consider

d04de75. The Retro-Thena team stated:

> - *Variable `TYPE` is just for UI and to recognize the contract's type from BscScan.*
> - *Variable `_VE` is just for having extra information in the gauge contract.*

Naming Suggestions

Several variables, parameters and functions throughout the codebase might benefit from better naming. Specifically:

- `VoterV3.sol`
- All occurrences of `_token` representing an array should be renamed to `_tokens`.
- All occurrences of `_poolVote` representing an array should be renamed to `_poolVotes`.
- `totWeightsPerEpoch` could be more clearly be `totalWeightsPerEpoch`.
- The `blacklist` functions could be renamed `removeFromWhitelist` for more clarity.
- `gaugesDistributionTimestmap` should be renamed `gaugeDistributionTimestmap`.
- `isAlive` could be `isGaugeAlive`.
- `isFactory` could be `isPairFactory`.

- `GaugeExtraRewarder.sol`

- `stop` could be more clearly named `paused`.

- `lastDistributedTime` could be `endOfDistributionPeriod` or something similar.
- Function `onReward` could be `claimReward`.
- The parameter `lpToken` of the `onReward` function could be `userBalance` or something similar.
- Function `setDistributionRate` could be `updateRewardRate`.
- Privileged entities `thenaMultisig` and `thenaTeamMultisig` could be given more descriptive names or docstrings that designate their distinct responsibilities.

- `GaugeV2.sol` and `GaugeV2_CL.sol`

- `Bribes.sol`

- `addRewards` could be named `addRewardTokens`. The same applies to functions `addReward` and `_addReward` in the same contract.

Update: *Partially resolved in pull request #6 at commit d720e4f. The Retro-Thena team stated:*

> *DISTRIBUTION: this is correct, the gauge is "standard" for any distributor and does not depend on voter.*

Non-explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease the clarity of the code, and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Throughout the codebase, global imports are being used.

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Acknowledged, not resolved.*

Typographical Errors

The following typographical errors have been discovered:

- Lines 221-225 in the `Bribes` contract`: "timestmap" should be "timestamp".
- Lines 111 and 117 in the `Bribes` contract: "of a owner" should be "of an owner".
- Lines 53, 763 and 773 in the `VoterV3` contract: "gaugesDistributionTimestmap" should be "gaugesDistributionTimestamp".
- Line 182 in the `GaugeV2_CL` contract: "sinle" should be "single".
- Line 146 in the `MinterUpgradeable` contract: function `calculate_rebate` should be `calculate_rebase`.

Update: *Resolved in pull request #6 at commits ecd3134 and d720e4f.*

functions and variables. Throughout the repository, publicly accessible functions and variables are incorrectly named with leading underscores. For instance:

- `IBribe.sol`
  - `function _deposit(uint amount, uint tokenId) external`
  - `function _withdraw(uint amount, uint tokenId) external`
- `Bribes.sol`
  - `function _deposit(uint256 amount, uint256 tokenId) external`
  - `function _withdraw(uint256 amount, uint256 tokenId) public`
  - `mapping(uint256 => uint256) public _totalSupply`
- `IVoter.sol`
  - `function _ve() external`
- `VoterV3.sol`
  - `function _init(address[] memory _tokens, address _permissionsRegistry, address _minter) external`
  - `function _factories() external`
  - `function _gaugeFactories() external`
  - `function _notifyRewardAmount(uint amount) external`
  - `function _epochTimestamp() public`
  - `address public _ve`
- `GaugeV2.sol`
  - `function _periodFinish() external`
  - `IERC20 public _VE`
  - `uint256 public _totalSupply`
  - `mapping(address => uint256) public _balances`
- `GaugeV2_CL.sol`
  - `function _periodFinish() external`
  - `IERC20 public _VE`
  - `uint256 public _totalSupply`
  - `mapping(address => uint256) public _balances`
- `PermissionsRegistry.sol`

Consider fixing all instances of prefixed underscores that conflict with the Solidity Style Guide.

Update: *Resolved in pull request #6 at commit 65d8936 and in pull request #8 at commit 200ba8b.*

Redefinition of Solidity Constants

In numerous places, hardcoded units of time are used rather than the native solidity constants for units of time.

The number `604800` is used instead of `1 weeks`:

- On line 689 of the `VoterV3` contract
- On line 835 of the `VoterV3` contract

The number `86400` is used instead of `1 days`:

- On line 878 of the `VoterV3` contract
- On line 31 of the `RewardsDistributor` contract
- On line 53 of the `GaugeExtraRewarder` contract
- On line 90 of the `GaugeV2` contract
- On line 94 of the `GaugeV2_VL` contract

Consider using Solidity's built-in numeric constants instead of hardcoded numbers in order to improve the clarity of the codebase.

Update: *Resolved in pull request #6 at commit 47aa201 and in pull request #11 at commit b36ad41.*

Mutable Variables Never Assigned or Only Assigned Once

The following variables in the `Bribes` contract are only assigned once and can be declared `immutable`:

- `bribeFactory`
- `ve`

- `DURATION`
- `TOKEN`
- `isForPair`
- `rewardToken`

The following variables in the `GaugeV2_CL` contract are only assigned once and can be declared `immutable`:

- `DURATION`
- `TOKEN`
- `rewardToken`

The following variables in the `GaugeExtraRewarder` contract are never assigned and can be declared `constant`:

- `ACC_TOKEN_PRECISION`
- `distributePeriod`

Consider declaring variables that are never assigned as `constant` and variables that are assigned only once as `immutable`.

Update: *Resolved in pull request #6 at commit 7626d97.*

Incorrect or Misleading Documentation

- On line 114 of the `GaugeV2_CL` contract, the comment reads "GaugeProxyL2" but the distribution address should be that of the `VoterV3` contract.
- On line 305 of the `VoterV3` contract, the comment reads "Revive a malicious gauge" but should read "Revive a killed gauge" as there is no reason to revive an actively malicious gauge.
- On lines 11 and 14 of the `PermissionsRegistry` contract, the comments suggest that `thenaMultisig` and `thenaTeamMultisig` have the same responsibilities and these responsibilities relate to this contract only, while neither is true.

Consider correcting the incorrect or misleading documentation.

Across the codebase, both `uint` and `uint256` keywords are used to declare unsigned integers of 256 bits in length.

To favor explicitness, consider standardizing all instances to `uint256`.

Update: *Partially resolved in pull request #6 at commit aa2a0f6 and in pull request #11 at commit 4b654a5. There are a few cases where `uint` is still used.*

Code Execution Continues When It Should Revert

In the `VoterV3` contract, both `attachTokenToGauge` and `detachTokenFromGauge` continue if the checks around `tokenId > 0` are not satisfied. In this case, the token is not attached and the overall operation fails.

Consider reverting in the case of a failed operation to better convey the failure to the user.

Update: *Resolved in pull request #6 at commit 51711b4. The functions `attachTokenToGauge` and `detachTokenFromGauge` have been removed completely.*

Uncalled External Functions

In the `VoterV3` contract, the functions `attachTokenToGauge` and `detachTokenFromGauge` require that the caller be a contract in the `isGauge` mapping. However, neither of the gauge implementations calls this function.

If these functions are unnecessary, consider removing them.

Update: *Resolved in pull request #6 at commit 51711b4.*

Client Reported

Emergency Mode Cannot Be Deactivated in Gauges

Because of a typographical error in the `stopEmergencyMode` function of the contracts `GaugeV2` and `GaugeV2_CL`, once the emergency mode has been activated it can no longer be deactivated.

The `setGaugeRewarder` function in the `GaugeV2` and `GaugeV2_CL` contracts does not allow setting the gauge rewarder to zero. However, `_deposit`, `_withdraw`, and the `_getReward` `functions` check the `gaugeRewarder` variable against `address(0)`. Since `address(0)` is used as a sentinel value to determine whether or not the extra rewarder is called, it should be possible to set the address to zero.

Update: *Resolved in pull request #7 at commit a5da2a2.*

One critical and four high-severity issues were found among various lower-severity issues. The system documentation and diagrams provided by the Thena Finance team made it easier for the auditors to assess and comprehend the code. Throughout the audit, the team has been responsive, providing insights and detailed explanations while engaging in the discussions with the auditors.

Several changes were suggested to better secure the system and ensure its functionality. Depending on the amount of code refactoring, a second audit may be recommended. We expect these changes to significantly improve the overall quality and maturity of the codebase.

While audits help in identifying code-level issues in the current implementation and potentially the code deployed in production, the Thena Finance team is encouraged to consider incorporating monitoring activities in the production environment. Ongoing monitoring of deployed contracts helps identify potential threats and issues affecting production environments. With the goal of providing a complete security assessment, this section raises several actions addressing trust assumptions and out-of-scope components that can benefit from on-chain monitoring.

Privileged entities and roles

Critical: There are numerous privileged actions with serious security implications as described in detail in the Security Model and Trust Assumptions section of this report. Consider monitoring triggers of all administrator functions to ensure all changes are expected. This should help the team remain vigilant against malicious actors.

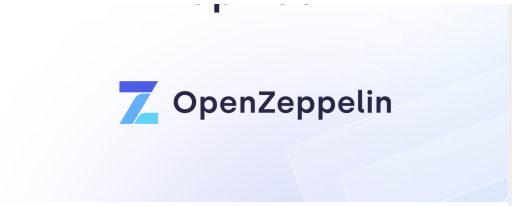
Technical

High: The credibility and accounting consistency of the system relies upon the timely actions of off-chain actors. Consider monitoring the voting period updates to validate that they take place exactly once per week. In addition, consider monitoring the distribution of the $THE emissions to the gauges to validate that it takes place once per voting period.

Suspicious Activity

Low: Consider monitoring users' interaction with the system, to track unusual activity such as depositing and withdrawing suspiciously large amounts instantly. This could help identify attempts to game the system.

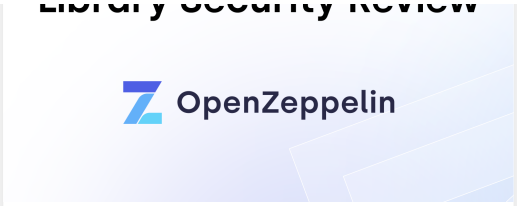
Related Posts



Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

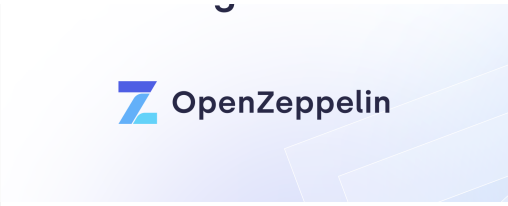
Security Audits



OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits



Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits