veRWA
Findings & Analysis Report

2023-10-11

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the veRWA smart contract system written in . The audit took place between August 7—August 10 2023.

Wardens

134 Wardens contributed reports to the veRWA :

1. ladboy233
2. bart1e
3. deadrxsezzz

4. OxDetermination
5. gzeon
6. Franfran
7. Yanchuan
8. catellatech
9. OxComfyCat
10. MrPotatoMagic
11. RED-LOTUS-REACH (BlockChomper, DedOhWale, SaharDevep, reentrant, and escrow)
12. rjs
13. popular00
14. cducrest
15. mert_eren
16. immeas
17. oakcobalt
18. SpicyMeatball
19. bin2chen
20. Brenzee
21. nonseodion
22. Team_Rocket (AlexCzm, and EllipticPoint)
23. ltyu
24. thekmj
25. OxCiphky
26. GREY-HAWK-REACH (Kose, aswinraj94, dimulski, aslanbek, and Oxraion)
27. pep7siup
28. Oxbrett8571
29. kaden
30. auditsea
31. markus_ether

32. Eeyore
33. 0x73696d616f
34. ppetrov
35. QiuhaoLi
36. 3docSec
37. 0xDING99YA
38. lsaudit
39. Jorgect
40. Tripathi
41. 0xSmartContract
42. zhaojie
43. qpzm
44. carrotsmuggler
45. nemveer
46. ADM
47. Yuki
48. Tendency
49. Tricko
50. th13vn
51. BenRai
52. KmanOfficial
53. Watermelon
54. lanrebayode77
55. seerether
56. Topmark
57. RandomUser
58. Kow
59. aakansha
60. Raihan

61. 0xkazim
62. d23e
63. ni8mare
64. jat
65. 0xhacksmithh
66. Shubham
67. klau5
68. AlexCzm
69. sandy
70. 14si2o_Flint
71. merlin
72. Deekshith99
73. windhustler
74. owadez
75. supervrijdag
76. carlos__alegre
77. hunter_w3b
78. kutugu
79. 0x3b
80. MatricksDeCoder
81. castle_chain
82. Bughunter101
83. Rolezn
84. ayden
85. 0x4non
86. audityourcontracts
87. Alhakista
88. imkapadia
89. InAllHonesty

90. leasowillow
91. Strausses
92. HChang26
93. _eperezok
94. erebus
95. deth
96. T1MOH
97. RHaO-sec
98. kaveyjoe
99. JP_Courses
100. Naubit
101. tay054
102. wahedtalash77
103. halden
104. Mike_Bello90
105. sl1
106. 0xStalin
107. devival
108. Bube
109. p_crypt0
110. 0xG0P1
111. 0xmuxyz
112. ch0bu
113. SUPERMAN_I4G
114. matrix_0wl
115. Giorgio
116. 0xE1
117. hassan-truscova
118. 0xweb3boy

119. fatherOfBlocks
120. Silverskrrrt
121. 0xWaitress
122. koxuan
123. piyushshukla
124. pipidu83
125. hpsb

This audit was judged by alcueca.

Final report assembled by PaperParachute.

Summary

The C4 analysis yielded an aggregated total of 11 unique vulnerabilities. Of these vulnerabilities, 8 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 96 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 veRWA repository, and is composed of 3 smart contracts written in the Solidity programming language and includes 749 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (8)

[H-01] User doesn't have to deposit for a week into the market to get their weekly reward from the `LendingLedger`

Submitted by SpicyMeatball, *also found by* mert_eren, nonseodion, cducrest, immeas, popular00, 0xComfyCat, GREY-HAWK-REACH, Yanchuan, ppetrov, kaden, *and* pep7siup

In the `LendingLedger` contract, a user is rewarded with CANTO tokens depending on how long he has his deposit in the market. Rewards are distributed for each week during which the deposit was inside the market. However, the user can cheat this condition because we are rounding down to the start of the week, so the user can deposit at 23:59 at the end of the week and withdraw at 00:00 and still get rewarded as if he had his deposit for the whole week.

Proof of Concept

▶ Details

Tools Used

Foundry

Recommended Mitigation Steps

It's difficult to propose a solution for this exploit without major changes in the contract's architecture. Perhaps we can somehow split the amount based on the time the sync was made inside the week, let's say Alice's `last_sync` was in the middle of week0, she deposited 1 ether, thus her amount for the current epoch will be 1/2 ether. However there is a caveat, how do we fill the gaps? We can't fill them with 1/2 ether. We can use this struct though,

```
Amount {
    uint256 actualAmount,
    uint256 fraction
}
```

so we can use `fraction` for the current epoch and `actualAmount = 1 ether` to fill the gaps.

alcueca (Judge) increased severity to High and commented:

> Chosen as best due to clarity, conciseness, and presence of executable PoC

> The rationale behind the High severity is that the purpose of veRWA is to attract liquidity to certain contracts as voted by CANTO holders, and this vulnerability defeats the purpose of attracting liquidity completely.

OpenCoreCH (veRWA) commented:

> Reward calculation is now based on a time-weighted balance. Btw, while implementing the fix I noticed that the PoC here does not really highlight the problem. In the PoC, there is only one lender, so even if we take the deposit time into account, this lender should receive 100% of the epoch rewards (as they provided 100% of the liquidity within the market during this epoch). I modified the PoC to a scenario where there are two lenders, with one that deposited only for one second and one for the whole week. The one that deposited for the whole week should receive ~604800 times more rewards for this epoch, which is now the case:

▶ Details

OpenCoreCH (veRWA) confirmed on duplicate finding #71

[H-02] Voters from VotingEscrow can vote infinite times in vote*for*gauge_weights() of GaugeController

Submitted by 0x73696d616f, *also found by* mert_eren, oakcobalt, SpicyMeatball, Tricko, 0xComfyCat, QiuhaoLi, Team_Rocket, Yanchuan, immeas, GREY-HAWK-REACH, th13vn, *0xCiphky (*1, 2*)*, ltyu, deadrxsezzz, nonseodion, lanrebayode77, 0xDetermination, popular00, *and* kaden

https://github.com/code-423n4/2023-08-verwa/blob/main/src/GaugeController.sol#L211
https://github.com/code-423n4/2023-08-verwa/blob/main/src/VotingEscrow.sol#L356

Delegate mechanism in `VotingEscrow` allows infinite votes in `vote_for_gauge_weights()` in the `GaugeController`. Users can then, for example, claim more tokens in the `LendingLedger` in the market that they inflated the votes on.

Proof of Concept

`VotingEscrow` has a delegate mechanism which lets a user delegate the voting power to another user. The `GaugeController` allows voters who locked native in `VotingEscrow` to vote on the weight of a specific gauge.

Due to the fact that users can delegate their voting power in the `VotingEscrow`, they may vote once in a gauge by calling `vote_for_gauge_weights()`, delegate their votes to another address and then call again `vote_for_gauge_weights()` using this other address.

A POC was built in Foundry, add the following test to `GaugeController.t.sol`:

▶ Details

Tools Used

Vscode, Foundry

Recommended Mitigation Steps

The vulnerability comes from the fact that the voting power is fetched from the current timestamp, instead of n blocks in the past, allowing users to vote, delegate, vote again and so on. Thus, the voting power should be fetched from n blocks in the past.

Additionaly, note that this alone is not enough, because when the current block reaches n blocks in the future, the votes can be replayed again by having delegated to another user n blocks in the past. The exploit in this scenario would become more difficult, but still possible, such as: vote, delegate, wait n blocks, vote and so on. For this reason, a predefined window by the governance could be scheduled, in which users can vote on the weights of a gauge, n blocks in the past from the scheduled window start.

alcueca (Judge) commented:

> Chosen as best due to the clear and concise explanation, including business impact on the protocol, and including an executable PoC.

OpenCoreCH (veRWA) confirmed on duplicate finding 86

[H-03] When adding a gauge, its initial value has to be set by an admin or all voting power towards it will be lost

Submitted by deadrxsezzz, *also found by* oakcobalt, 0xComfyCat, Yanchuan, Brenzee, bin2chen, auditsea, cducrest, markus_ether, *and* Team_Rocket

https://github.com/code-423n4/2023-08-verwa/blob/main/src/GaugeController.sol#L118

https://github.com/code-423n4/2023-08-verwa/blob/main/src/GaugeController.sol#L204

Voting power towards gauges will be lost and project will not work properly

Proof of Concept

The mapping `time_weight` takes a gauge as a param and returns the most recent timestamp a gauge has had its weight recorded/ updated. There are 2 ways to set this value: through `_get_weight` and `_change_gauge_weight`.

▶ Details

Tools Used

Foundry

Recommended Mitigation Steps

Upon adding a gauge, make a call to `change_gauge_weight` and set its initial weight to 0.

__141345__ (Lookout) commented:

> Forget to initialize `time_weight[]` when add new gauge.

[H-04] Delegated votes are locked when owner lock is expired

Submitted by ltyu, *also found by* qpzm, RED-LOTUS-REACH, bart1e, 0xDING99YA, zhaojie, popular00, MrPotatoMagic, carrotsmuggler, pep7siup, 3docSec, mert_eren, kaden, Yuki, seerether, KmanOfficial, *cducrest (*1, 2*)*, Tendency, *and* bin2chen

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L331

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L371-L374

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L383

In `delegate()` of VoteEscrow.sol, a user is able to delegate their locked votes to someone else, and undelegate (i.e. delegate back to themselves). When the user tries to re-delegate, either to someone else or themselves, the lock must not be expired. This is problematic because if a user forgets and lets their lock become expired, they cannot undelegate. This blocks withdrawal, which means their tokens are essentially locked forever.

Proof of Concept

To exit the system, Alice must call `withdraw()`. However, since they've delegated, they will not be able to.

▶ Details

Recommended Mitigation Steps

Consider refactoring the code to skip `toLocked.end > block.timestamp` if undelegating. For example, adding a small delay (e.g., 1 second) to the lock end time when a user undelegates.

alcueca (Judge) commented:

> This vulnerability, if not found, would have meant that some users would have permanently lost assets in the form of voting power. While at that point the application owners would certainly warn users to not let their locks expire without undelegating, many users would not get the warning, as it is not that easy to make sure that every user is aware of something. The result is that time and again, users would get their tokens locked forever.

OpenCoreCH (veRWA) confirmed on duplicate 112

[H-05] It is possible to DoS all the functions related to some gauge in `GaugeController`

Submitted by bart1e*, also found by* 0xDetermination

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeControll er.sol#L91-L114

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L142

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L180

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L189

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L247

`_get_weight` function is used in order to return the total gauge's weight and it also updates past values of the `points_weight` mapping, if `time_weight[_gauge_addr]` is less or equal to the `block.timestamp`. It contains the following loop:

```
        for (uint256 i; i < 500; ++i) {
            if (t > block.timestamp) break;
            t += WEEK;
            uint256 d_bias = pt.slope * WEEK;
            if (pt.bias > d_bias) {
                pt.bias -= d_bias;
                uint256 d_slope = changes_weight[_gauge_addr
                pt.slope -= d_slope;
            } else {
                pt.bias = 0;
                pt.slope = 0;
            }
            points_weight[_gauge_addr][t] = pt;
            if (t > block.timestamp) time_weight[_gauge_addr
        }
```

There are two possible scenarios:

- `pt.bias > d_bias`
- `pt.bias <= d_bias`

The first scenario will always happen naturally, since `pt.bias` will be the total voting power allocated for some point and since slope is a sum of all users' slopes and slopes are calculated in such a way that `<SLOPE> * <TIME_TO_END_OF_STAKING_PERIOD> = <INITIAL_BIAS>`.

However, it is possible to artificially change `points_weight[_gauge_addr][t].bias` by calling `change_gauge_weight` (which can be only called by the governance). It important to notice here, that `change_gauge_weight` **doesn't modify** `points_weight[_gauge_addr][t].slope`

`change_gauge_weight` does permit to change the weight to a smaller number than its current value, so it's both perfectly legal and possible that governance does this at some point (it could be changing the weight to `0` or any other value smaller than the current one).

Then, at some point when `_get_weight` is called, we will enter the `else` block because `pt.bias` will be less than the sum of all user's biases (since originally these values were equal, but `pt.bias` was lowered by the governance). It will set `pt.bias` and `pt.slope` to `0`.

After some time, the governance may realise that the gauge's weight is `0`, but should be bigger and may change it to some bigger value.

We will have the situation where `points_weight[_gauge_addr][t].slope = 0` and `points_weight[_gauge_addr][t].bias > 0`.

If this happens and there is any nonzero `changes_weight[_gauge_addr]` not yet taken into account (for instance in the week after the governance update), then all the functions related to the gauge at `_gauge_addr` will not work.

It's because, the following functions:

- `checkpoint_gauge`
- `gauge_relative_weight_write`
- `gauge_relative_weight`
- `_change_gauge_weight`
- `change_gauge_weight`
- `vote_for_gauge_weights`
- `remove_gauge`

call `_get_weight` at some point.

Let's see what will happen in `_get_weight` when it's called:

```
uint256 d_bias = pt.slope * WEEK;
if (pt.bias > d_bias) {
    pt.bias -= d_bias;
    uint256 d_slope = changes_weight[_gauge_addr
    pt.slope -= d_slope;
} else {
```

We will enter the `if` statement, because `pt.bias` will be `> 0` and `pt.slope` will be `0` (or some small value, if users give their voting power to gauge in the meantime), since it was previously set to `0` in the `else` statement and wasn't touched when gauge's weight was changed by the governance. We will:

- Subtract `d_bias` from `pt.bias` which will succeed
- Attempt to subtract `changes_weight[_gauge_addr][t]` from `d_slope`

However, there could be a user (or users) whose voting power allocation finishes at `t` for some `t` not yet handled. It means that `changes_weight[_gauge_addr][t] > 0` (and if `pt.slope` is not `0`, then `changes_weight[_gauge_addr][t]` still may be greater than it).

If this happens, then the integer underflow will happen in `pt.slope -= d_slope;`. It will now happen in **every** call to `_get_weight` and it won't be possible to recover, because:

- `vote_for_gauge_weights` will revert
- `change_gauge_weight` will revert

as they call `_get_weight` internally. So, it won't be possible to modify `pt.slope` and `pt.bias` for any point in time, so the `revert` will always happen for that gauge. It won't even be possible to remove that gauge.

So, in short, the scenario is as follows:

1. Users allocate their voting power to a gauge `X`.
2. Governance at some point decreases the weight of `X`.
3. Users withdraw their voting power as the time passes, and finally the weight of `X` drops to `0`.
4. Governance realises this and increases weight of `X` since it wants to incentivise users to provide liquidity in `X`.
5. Voting power delegation of some user(s) ends some time after that and `_get_weight` attempts to subtract `changes_weight[_gauge_addr][t]` from the current slope (which is either `0` or some small value) and it results in integer underflow.
6. `X` is unusable and it's impossible to withdraw voting power from (so users cannot give their voting power somewhere else). The weight of `X` cannot be changed anymore and `X` cannot be even removed.

Note that it is also possible to frontrun the call to `change_gauge_weight` when the weight is set to a lower value - user with a lot of capital can watch the mempool and if weight is lowered to some value `x`, he can give a voting power of `x` to that gauge. Then, right after weight is changed by the governance, he can withdraw his voting power, leaving the gauge with weight = `0`. Then, governance will manually increase the weight to recover and DoS will happen as described. **So it is only needed that governance decreases gauge's weight at some point.**

Impact

As stated, above the impact is that the entire gauge is useless, voting powers are permanently locked there and its weight is impossible to change, so the impact is high.

In order for this situation to succeed, governance has to decrease weight of some gauge, but I think it's very likely, because:

1. `_get_weight` checks that `if (pt.bias > d_bias)` and it handles the opposite situation, so it is anticipated that it may genuinely happen.
2. It is definitely possible to decrease gauge's weight and it's even possible to zero it out (as in the `remove_gauge`).
3. The situation where `old_bias` is greater than `old_sum_bias + new_bias` is handled in `vote_for_gauge_weights`, but it may only happen when gauge's weight was decreased by the governance.
4. The situation where `old_slope.slope` is greater than `old_sum_slope + new_slope.slope` is also handled there, but it may only happen if we enter the `else` statement in `_get_weight`.

So, it is predicted that gauge's weight may be lowered and the protocol does its best to handle it properly, but as I showed, it fails to do so. Hence, I believe that this finding is of High severity, because although it requires governance to perform some action (decrease weight of some gauge), I believe that it's likely that governance decides to decrease weight, especially that it is anticipated in the code and edge cases are handled there (and they wouldn't be if we assumed that governance would never allowed them to happen).

Proof of Concept

Please run the test below. The test shows slightly simplified situation where governance just sets weight to `0` for `gauge1`, but as I've described above, it suffices that it's just changed to a smaller value and it may drop to `0` naturally as users withdraw their voting power. The following import will also have to be added: `import {Test, stdError} from "forge-std/Test.sol";`.

▶ Details

Tools Used

VS Code

Recommended Mitigation Steps

Perform `pt.slope -= d_slope` in `_get_weight` only when `pt.slope >= d.slope` and otherwise zero it out.

__141345__ (Lookout) commented:

> `pt.slope -= d_slope` underflow, DoS gauge operation.

OpenCoreCH (veRWA) confirmed

alcueca (Judge) commented:

> This finding does a great job at describing the vulnerability and its impact from a computational point of view, including an executable PoC. Its duplicate **#386** is also worthy of note since it explains the root cause from a mathematical point of view. Although this finding was selected as best, both findings should be read for their complementary points of view.

[H-06] Users may be forced into long lock times to be able to undelegate back to themselves

Submitted by **ADM***, also found by Isaudit (1, 2),* **QiuhaoLi***, Jorgect (1, 2),* **SpicyMeatball***, bart1e (1, 2),* **Yanchuan***,* **3docSec***,* **MrPotatoMagic***, nemveer (1, 2),* **Yuki***,* **kaden***, nonseodion (1, 2),* **Watermelon***,* **RandomUser***, BenRai (1, 2),* **cducrest***,* **Topmark***,* **Tendency***,* **0xDING99YA***, and* **Kow**

Due to a check requiring users only be able to delegate to others or themselves with longer lock times and verwa's restrictions of all changes increasing lock times by 5 years users may be forced to remain delegated to someone they do not wish to be or extend their lock much longer than they wish.

Proof of Concept

If a user does not delegate to another user who started their lock during the same epoch they will be unable to undelegate back to themselves without extending their own lock. This is not much of an issue if they wish to do so early in the lock period but can become a problem if they wish to delegate to themselves after a longer period of time. i.e.

1. Bob creates lock in week 1.
2. Dave create lock in week 2 & Bob delegates to Dave.
3. 3 years pass and Bob decides he wishes to undelegate his votes back to himself and calls delegate(msg.sender) but the call will fail due to the check in VotingEscrow#L384:

```
require(toLocked.end >= fromLocked.end, "Only delegate to longer
```

4. In the original FiatDAO contracts a user would be able to just extend their lock by one week to get around this however any changes in the verwa contract results in an extension of 5 years which the user may not want extend their lock by such a long time just to be able to undelegate.

The undelegate fail can be shown by modifying the test testSuccessDelegate to:

```
    function testSuccessDelegate() public {
        // successful delegate
        testSuccessCreateLock();
        vm.warp(8 days); // warp more than 1 week so both users
        vm.prank(user2);
        ve.createLock{value: LOCK_AMT}(LOCK_AMT);
        vm.prank(user1);
        ve.delegate(user2);
        (, , , address delegatee) = ve.locked(user1);
        assertEq(delegatee, user2);
        (, , int128 delegated, ) = ve.locked(user2);
        assertEq(delegated, 2000000000000000000);
    }
```

and running:
forge test `--match testSuccessUnDelegate`

Recommended Mitigation Steps

Modify VotingEscrow#L384 to:

```
require(toLocked.end >= locked_.end, "Only delegate to self or l
```

which will allow users to delegate either to longer locks or undelegate back to themselves.

alcueca (Judge) increased severity to High and commented:

> I'm merging #245 into this one as the root cause and general mechanics are the same, only that in the 245 group the intent was malicious and in this group is not.

> At the same time, I'm upgrading the severity to High. Locking CANTO for an additional 5 years, considering that this is by nature a volatile environment, has an extremely high chance of resulting in losses due to market movements or other factors.

OpenCoreCH (veRWA) confirmed on duplicate 178

[H-07] lack of access control in `LendingLedger.sol#checkpoint_lender` and function `checkpoint_market`

Submitted by ladboy233

Note: This audit was preceded by a Code4rena Test Coverage competition*. While auditing was not the purpose of the testing phase, relevant and valuable findings reported during that phase were eligible to be judged. This finding [H-07] was discovered during that period and is being included here for completeness.*

User's claim and `sync_lender` will be griefed at low cost.

Proof of Concept

▶ Details

These two functions lack access control, the caller is never validated, meaning anyone can call this function.

The market is not validated to see if the market is whitelisted or not.

The timestamp is never validated, the is_valid_epoch(_forwardTimestampLimit) is insufficient.

```
    /// @notice Check that a provided timestamp is a valid epoch (
      /// @param _timestamp Timestamp to check
      modifier is_valid_epoch(uint256 _timestamp) {
          require(
              _timestamp % WEEK == 0 || _timestamp == type(uint256
              "Invalid timestamp"
          );
          _;
      }
```

The user can just pick a past timestamp as the _forwardTimestampLimit.

For example, if we set _forwardTimestampLimit to 0.

Then for example in _checkpoint_market

```
     function _checkpoint_market(
          address _market,
          uint256 _forwardTimestampLimit
      ) private {
          uint256 currEpoch = (block.timestamp / WEEK) * WEEK;
          uint256 lastMarketUpdateEpoch = lendingMarketTotalBalanc
          uint256 updateUntilEpoch = Math.min(currEpoch, _forwardT
          if (lastMarketUpdateEpoch > 0 && lastMarketUpdateEpoch <
              // Fill in potential gaps in the market total balanc
              uint256 lastMarketBalance = lendingMarketTotalBalanc
                  lastMarketUpdateEpoch
              ];
              for (
                  uint256 i = lastMarketUpdateEpoch;
                  i <= updateUntilEpoch;
                  i += WEEK
              ) {
                  lendingMarketTotalBalance[_market][i] = lastMark
              }
          }
          lendingMarketTotalBalanceEpoch[_market] = updateUntilEpo
      }
```

We set the lendingMarketTotalBalanceEpoch[_market] to 0.

Then if the next call of the _checkpoint_market, the for loop would never run because the lastMarketUpdateEpoch is 0.

Over time, even when the for loop inside _checkpoint_market does run, the caller are forced to pay very high gas fee.

Same issue applies to _checkpoint_lender as well.

User can decrease lendingMarketBalancesEpoch, even to 0.

Basically, if a malicious actor call these two function with forwardTimestampLimit 0.

Then the _checkpoint_lender and _checkpoint*market would never run inside sync*\ledger and claim reward.

Because user's reward can be griefed to 0 and stated are failed to updated properly.

POC 1:

▶ Details

If we run the POC, we get the normal result, user can claim and get 6 ETH as reward.

```
---sync ledger after set the update epoch to 0 --
6000000000000000000
```

If we uncomment:

```
 // ledger.checkpoint_market(lendingMarket, 0);
 // ledger.checkpoint_lender(lendingMarket, lender, 0);
```

The claimed reward goes to 0.

Recommended Mitigation Steps

Add access control to checkpoint_market and checkpoint_lender.

[OpenCoreCH (veRWA) confirmed and commented](#):

> This was a valid issue before the auditing contest (uncovered during the testing contest and fixed before the auditing contest), pasting my comment from there for reference:

> Good point. It is generally intended that everyone can call these functions (should not be necessary in practice, but may be in some edge cases where a market was inactive for years) and I do not think that this is problematic per se. However, the problem here is that users can decrease `lendingMarketBalancesEpoch` or `lendingMarketTotalBalanceEpoch`, which should never happen. So I will probably change the code like this (and the same for lenders) such that this can never happen:

```
        if (lastMarketUpdateEpoch > 0 && lastMarketUpdateEpoch <
            // Fill in potential gaps in the market total balanc
            uint256 lastMarketBalance = lendingMarketTotalBalanc
            for (uint256 i = lastMarketUpdateEpoch; i <= updateU
                lendingMarketTotalBalance[_market][i] = lastMark
            }
            if (updateUntilEpoch > lastMarketUpdateEpoch) {
                lendingMarketTotalBalanceEpoch[_market] = update
            }
        }
```

[H-08] If governance removes a gauge, user's voting power for that gauge will be lost

Submitted by [thekmj](#), *also found by* [mert_eren](#), [popular00](#), [Eeyore](#), [immeas](#), [bart1e](#), [0xCiphky](#), [ltyu](#), [0xbrett8571](#), [deadrxsezzz](#), [0xDetermination](#), [Tripathi](#), [Team_Rocket](#), *and* [pep7siup](#)

https://github.com/code-423n4/2023-08-verwa/blob/main/src/GaugeController.sol#L127-L132 https://github.com/code-423n4/2023-08-verwa/blob/main/src/GaugeController.sol#L213

https://github.com/code-423n4/2023-08-verwa/blob/main/src/GaugeController.sol#L241

If governance removes a gauge for any (non-malicious) reason, a user's voting power for that gauge will be completely lost.

Vulnerability details

The `GaugeController` is a solidity port of Curve DAO's Vyper implementation. Users are to vote for channeling incentives by using the `vote_for_gauge_weights()` function, and each user can fraction their voting power by 10000 (that is, defined by BPS).

One modification from the original is that governance can now remove gauges, not allowing users to vote on it. However, any existing individual user's voting power before removal is not reset. Since `vote_for_gauge_weights()` does not allow voting for removed gauges, the voting power is then forever lost.

Consider the following scenario:

- Alice has some veRWA, and is now able to vote.
- She votes on some pool, say, G1, using 100% of her voting power.
- Pool G1 is removed by governance due to any reason. Perhaps the pool was found to be faulty and liquidity should be migrated, perhaps the market itself has became illiquid and unsafe, perhaps the intended incentives duration for that pool has simply expired.
- Alice still has 100% of her voting power in that pool, but she cannot remove her vote and claim the voting power back.

It is worth noting that, even if Alice does not use 100% of her voting power on that particular gauge, she would still lose whatever percent vote placed in that pool, and her overall voting power was weakened by said percent.

Impact

Users can lose their voting power.

Proof of concept

We provide the following POC to use on `GaugeController` tests.

▶ Details

Tools used

Forge

Recommended mitigation steps

The simplest way to mitigate this is to **allow zero-weight votings on expired pools** simply to remove the vote. Modify line 213 as follow:

```
    require(_user_weight == 0 || isValidGauge[_gauge_addr], "Can onl
```

https://github.com/code-423n4/2023-08-verwa/blob/main/src/GaugeController.sol#L213

The given POC can then be the test case to verify successful mitigation.

As a QA-based recommendation, the sponsor can also provide an external function to remove votes, and/or provide a function to vote for various pools in the same tx. This will allow users to channel their votes directly from removed pools to ongoing pools.

Medium Risk Findings (3)

[M-01] Users can front-run calls to `change_gauge_weight` to gain extra voting power

Submitted by deadrxsezzz

Users can get extra voting power by front-running calls to `change_gauge_weight`.

Proof of Concept

It can be expected that in some cases calls will be made to `change_gauge_weight` to increase or decrease a gauge's weight. The problem is users can be monitoring the mempool expecting such calls. Upon seeing such, any people who have voted for said gauge can just remove their vote prior to `change_gauge_weight`. Once it executes, they can vote again for their gauge, increasing its weight more than it was expected to be: Example:

1. Gauge has 1 user who has voted and contributed for 10_000 weight
2. They see an admin calling `change_gauge_weight` with value 15_000.
3. User front-runs it and removes all their weight. Gauge weight is now 0.
4. Admin function executes. Gauge weight is now 15_000
5. User votes once again for the gauge for the same initial 10_000 weight. Gauge weight is now 25_000.

Gauge weight was supposed to be changed from 10_000 to 15_000, but due to the user front-running, gauge weight is now 25_000.

Recommended Mitigation Steps

Instead of having a set function, use increase/ decrease methods.

alcueca (Judge) commented:

> Votes are only counted at the first second of an epoch, front-running `change_gauge_weight` won't do anything.

deadrxsezzz (Warden) commented:

> Hey, I'm leaving a simple PoC showcasing the issue and the respective logs from it:

▶ Details

OpenCoreCH (veRWA) commented:

> In practice the function should only be used to set the weights to 0 (e.g., when a market got exploited and not all users withdrew their votes yet). But there is

> nothing that prevents it from setting it to a higher or lower value (at least not in the audited codebase, added that in the meantime). So what is described in the finding is generally true, although it does not require frontrunning or anything like that, because the function just changes the current voting result, anyway. So if voting for the market is not restricted afterwards (by removing it from the whitelist), people can continue to vote on it and the final vote result may be different than what governance has set with this function.

alcueca (Judge) commented:

> From what I understand the function is essentially broken, and a Medium severity is appropriate given that wardens can't assume that it will be used only to set the weights to 0 for non-whitelisted markets.

OpenCoreCH (veRWA) confirmed on duplicate 401

[M-02] Upon IncreaseAmount the lock may not align to the nearest weekly increment

Submitted by gzeon

Note: This audit was preceded by a Code4rena Test Coverage competition*. While auditing was not the purpose of the testing phase, relevant and valuable findings reported during that phase were eligible to be judged. This finding [M-02] was discovered during that period and is being included here for completeness.*

In most other places, locks are created with the end time rounded down to nearest weekly increment using the `_floorToWeek` function.

https://github.com/OpenCoreCH/test-squad-verwa/blob/461b7d99a30e8fee57ba97c2262f6b0c5e4704b6/src/VotingEscrow.sol#L422-L426

```
    // @dev Floors a timestamp to the nearest weekly increment
    // @param _t Timestamp to floor
    function _floorToWeek(uint256 _t) internal pure returns (uir
        return (_t / WEEK) * WEEK;
    }
```

However in IncreaseAmount, it is simply set to `block.timestamp + LOCKTIME` without the rounding:

https://github.com/OpenCoreCH/test-squad-verwa/blob/461b7d99a30e8fee57ba97c2262f6b0c5e4704b6/src/VotingEscrow.sol#L301-L302

```
            newLocked.end = block.timestamp + LOCKTIME;
```

[__141345__ (Lookout) commented]():

> Locked.end will affect the delegate/undelegate requirement, such as the following check.

```
File: src\VotingEscrow.sol
382:         require(toLocked.amount > 0, "Delegatee has no lock
383:         require(toLocked.end > block.timestamp, "Delegatee
384:         require(toLocked.end >= fromLocked.end, "Only deleg
```

[OpenCoreCH (veRWA) confirmed and commented]():

> This was discovered during the testing contest and fixed before the audit contest, pasting my comment here for reference:

> Good catch, I think that would mess up the `_checkpoint` logic which implicitly assumes that the lock times are aligned, so will definitely be fixed.

[alcueca (Judge) commented]():

> Impact from sponsor:

> These slope changes would not have been applied (neither the user-specific ones, nor the global ones), resulting in wrong slopes. And if many users did this, many slopes would be messed up at some point, leading to wrong votes / relative votes.

[M-03] Replace `old_sum_bias` by `old_bias`

Submitted by Franfran

Note: This audit was preceded by a Code4rena Test Coverage competition. While auditing was not the purpose of the testing phase, relevant and valuable findings reported during that phase were eligible to be judged. This finding [M-03] was discovered during that period and is being included here for completeness.

```
diff --git a/src/GaugeController.sol b/src/GaugeController.sol
index 68b832a..1794639 100644
--- a/src/GaugeController.sol
+++ b/src/GaugeController.sol
@@ -250,7 +250,7 @@ contract GaugeController {
         uint256 old_sum_slope = points_sum[next_time].slope;

         points_weight[_gauge_addr][next_time].bias = Math.max(o
-        points_sum[next_time].bias = Math.max(old_sum_bias + ne
+        points_sum[next_time].bias = Math.max(old_sum_bias + ne
         if (old_slope.end > next_time) {
             points_weight[_gauge_addr][next_time].slope =
                 Math.max(old_weight_slope + new_slope.slope, ol
```

OpenCoreCH (veRWA) confirmed and commented:

> This was discovered during the testing contest and fixed before the auditing contest.

alcueca (Judge) commented:

> @OpenCoreCH, since the warden didn't really submit a report, would you be so kind as to explain the impact of this bug?

OpenCoreCH (veRWA) commented:

> The `Math.max` there is an underflow protection for `points_sum[next_time]`. This wrong implementation would have lead to an underflow in some edge cases (`points_sum` is near 0 / low, i.e. there is not a lot of voting power in the system), preventing votes for the user. Because `old_bias` decreases over time (and

> eventually reaches 0), the error would generally have been recoverable, but it could have taken some time.

Low Risk and Non-Critical Issues

For this audit, 96 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **RED-LOTUS-REACH** received the top score from the judge.

The following wardens also submitted reports: ppetrov, nemveer, aakansha, Raihan, 0xkazim, kaden, oakcobalt, popular00, d23e, ni8mare, jat, Eeyore, nonseodion, 0xDetermination, 0xhacksmithh, Shubham, rjs, klau5, AlexCzm, sandy, Watermelon, QiuhaoLi, auditsea, 14si2o_Flint, merlin, deadrxsezzz, Deekshith99, immeas, windhustler, owadez, supervrijdag, KmanOfficial, carlos__alegre, hunter_w3b, kutugu, 0x3b, MatricksDeCoder, castle_chain, Bughunter101, Rolezn, 0xCiphky, 0xDING99YA, ayden, 0x4non, MrPotatoMagic, audityourcontracts, ladboy233, Alhakista, imkapadia, InAllHonesty, leasowillow, Strausses, HChang26, _eperezok, erebus, deth, T1MOH, thekmj, RHaO-sec, kaveyjoe, JP_Courses, Naubit, tay054, wahedtalash77, halden, Mike_Bello90, Tripathi, sl1, lanrebayode77, 0xStalin, devival, Bube, p_crypt0, 0xG0P1, markus_ether, cducrest, zhaojie, 0xmuxyz, ch0bu, 0xbrett8571, lsaudit, SUPERMAN_I4G, matrix_0wl, Topmark, Giorgio, 0xE1, hassan-truscova, 0xweb3boy, fatherOfBlocks, Silverskrrrt, 0xWaitress, koxuan, piyushshukla, pipidu83, *and* hpsb.

[01] Casting between types makes values negative for huge input values

Invariants to the VotingEscrow contract can be broken. While an enormous quantity is necessary, greater than the total supply of `$CANTO`, it's worth mentioning due to the fact that invariants in the system can be broken, if only in testing environments.

Invariants affected in `VotingEscrow.sol`:

- `LockedBalance.delegated` must always be > 0
    - can be made negative

- `LockedBalance.amount` must always be > 0
  - can be made negative
- `Point.slope` must always be ≥ 0
  - can be made negative
- `Point.bias` must always be ≥ 0
  - can be made negative

Cases in `VotingEscrow.sol` where casting creates vulnerable attack surfaces:

- From increaseAmount in VotingLock.sol, line 317:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L317

```
newLocked.delegated += int128(int256(_value));
```

- From increaseAmount in VotingLock.sol, line 305:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L305

```
newLocked.delegated += int128(int256(_value));
```

- From createLock in VotingLock.sol, line 276:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L276

```
locked_.amount += int128(int256(_value));
```

- From createLock in VotingLock.sol, line 278:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L278

```
    locked_.delegated += int128(int256(_value));
```

- From increaseAmount in VotingLock.sol, line 300:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L300

```
    newLocked.amount += int128(int256(_value));
```

- From increaseAmount in VotingLock.sol, line 317:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L317

```
    newLocked.delegated += int128(int256(_value));
```

Risk

Debugging and auditing this code is made more complex due to the inclusion of this casting, which puts the protocol more at risk than had they been unused.

It's not fully clear why the types for `amount` and `delegated` of the `LockedBalance` struct are of type `int128`. This could introduce unforseen edge cases due to the copious casting. All assignments from function arguments are of type `uint256` (coming from `createLock()` and `increaseAmount()`), making the usage more peculiar and inconsistent. Similarly, `int128` is used for `bias` and `slope` of the `Point` struct, and in all places where they are used for calculations, they must be checked for being negative and are subsequently reset to `0` in the case of being negative. The suspicion is that since the origin of the `ve` system came from the original Curve contracts, the implicit assumptions of that system were kept wholesale.

Mitigation

Use unsigned integers for all financial calculations so that all casting is removed and remove the subsequent unnecessary "resets" to `0`. Instead, for places where the "resets" are used, define pre-conditions that ensure negative-valued outcomes are not permissible in calls to `createLock()` and `increaseLock()`.

[02] Bias and slope can be made negative with huge deposits thus manipulating total voting power

Similar to the previous "Casting between types makes values negative for huge input values", huge inputs can allow for attacks on the protocol, if only in a testing environment.

In `VotingEscrow.sol` the code in question is from `_checkpoint()`, lines 129 - 136:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L129-L136

```
    // When a huge deposit is made, the casting of _value in createI
    if (_oldLocked.end > block.timestamp && _oldLocked.delegated > (
        // ...these can become negative as a result
        userOldPoint.slope = _oldLocked.delegated / int128(int256(LC
        userOldPoint.bias = userOldPoint.slope * int128(int256(_oldI
    }
    if (_newLocked.end > block.timestamp && _newLocked.delegated > (
        // ...these can become negative as a result
        userNewPoint.slope = _newLocked.delegated / int128(int256(LC
        userNewPoint.bias = userNewPoint.slope * int128(int256(_newI
    }
```

Risk

Control over the sign of `bias` and `slope` allows for the control of decay of voting power, because at any point they become < 0, they are reset to 0, giving the theoretical attacker the ability to temporarily stem the decay of their voting power.

Mitigation

Similar to "Casting between types makes values negative for huge input values", bypass the need to cast (and automatically hard-lock values to 0) by using unsigned types with guards for relevant pre-conditions that ensure negative-valued outcomes are not permissible in calls to `createLock()` and `increaseLock()`.

[03] Inconsistent Lock Durations

Locks for veRWA are set for 5 years, given no further actions on a lock are performed. This 5 year period is an important duration, because it also determines the calculations for the decay in voting power a lock has. In `**VotingEscrow.sol**`, the 5 year lock period is measured as 1825 days.

Places where the 1785 day measurement is used:

- in `_checkpoint()`, line 185

```
for (uint256 i = 0; i < 255; i++) {
        iterativeTime = iterativeTime + WEEK;

        ...

        if (iterativeTime == block.timestamp) {
            lastPoint.blk = block.number;
            break;
        } ...
}
```

- in `_supplyAt()`, line 538

```
function _supplyAt(..., uint256 _t) ... {
        for (uint256 i = 0; i < 255; i++) {
                iterativeTime = iterativeTime + WEEK;
```

```
                    ...

                    if (iterativeTime == _t) {
                        break;
                    }
                }
        }
```

Risk

There is a possibility that the protocol faces a case where locked funds are untouched and no new funds are locked across a 5 year period. In this “last users standing” scenario, the protocol theoretically is still responsible for calculating the final voting weights, allowing investors to maximize the utility of their locked funds regardless of the relevancy of the protocol.

Surprisingly, the maximum span the protocol can actually measure is less than 5 years, it is 255 weeks, which equates to 1785 days. For those final 40 days and afterwards, for example, weights will be miscalculated, and, especially depending on the size of the funds locked, returns on those funds or supply totals will be miscalculated.

Mitigation

Increase the number of iterations to fully calculate weights through the 5 year period.

[04] Unnecessary Modifiers Cost Gas

Reentrancy is a common root-cause for exploits in smart contracts. To avoid the specific case of basic reentrancy, a `nonReentrant` modifier is often used. In the case of a handful of functions in `VotingEscrow.sol` , this modifier is unnecessary, because the cause of reentrancy - calling into external contracts - is also missing. While they are not a risk, they do cause extra computation, and thus raise gas costs, which may be of some notice especially during high network congestion.

Locations where they exist:

- Line 268:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L268

```
    function createLock(uint256 _value) external payable nonReentrar
```

- Line 288:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L288

```
    function increaseAmount(uint256 _value) external payable nonReer
```

- Line 326:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L326C5-L326

```
    function withdraw() external nonReentrant {
```

- Line 356:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L356

```
    function delegate(address _addr) external nonReentrant {
```

Risk

No identified risk, since no external contract calls are made in these functions. However there is the cost of computation from running the modifiers during each call, thus increasing gas costs.

Mitigation

Remove the modifiers from the function definitions.

[05] Many cases of precision loss from multiply-after-divide

While there wasn't enough time to explore possible attack vectors for these cases, it's apparent that there is definite loss in precision while calculating important values in the system.

- From `_checkpoint()` in `VotingEscrow.sol`, lines 129 - 132:

https://github.com/code-423n4/2023-08-verwa/blob/9a2e7be003bc1a77b3b87db31f3d5a1bcb48ed32/src/VotingEscrow.sol#L129-L132

```
    if (_oldLocked.end > block.timestamp && _oldLocked.delegated > (
        userOldPoint.slope = _oldLocked.delegated / int128(int256(LO
        userOldPoint.bias = userOldPoint.slope * int128(int256(_oldI
    }
```

- From `_checkpoint()` in `VotingEscrow.sol`, lines 133 - 136:

https://github.com/code-423n4/2023-08-verwa/blob/9a2e7be003bc1a77b3b87db31f3d5a1bcb48ed32/src/VotingEscrow.sol#L133-L136

```
    if (_newLocked.end > block.timestamp && _newLocked.delegated > (
            userNewPoint.slope = _newLocked.delegated / int128(int25
            userNewPoint.bias = userNewPoint.slope * int128(int256(_
    }
```

- From `_checkpoint()` in `VotingEscrow.sol`, lines 176 - 218

https://github.com/code-423n4/2023-08-verwa/blob/9a2e7be003bc1a77b3b87db31f3d5a1bcb48ed32/src/VotingEscrow.sol#L176-L218

```
    uint256 blockSlope = 0; // dblock/dt
    if (block.timestamp > lastPoint.ts) {
        blockSlope = (MULTIPLIER * (block.number - lastPoint.blk)) /
    }

    // Go over weeks to fill history and calculate what the current
    uint256 iterativeTime = _floorToWeek(lastCheckpoint);
    for (uint256 i = 0; i < 255; i++) {
         ...

        // PRECISION LOSS HERE: blockslope was originally calculatec
        lastPoint.blk = initialLastPoint.blk + (blockSlope * (iterat

         ...
    }
```

- From `_checkpoint_lender()` in LendingLedger.sol, line 60:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L60

```
    uint256 currEpoch = (block.timestamp / WEEK) * WEEK;
```

Risk

Precision loss in critical calculations can make the risk/reward for investors less appealing. In critical cases there could be possible losses involved when these discrepancies are exploited.

Mitigation

Reorder operations to ensure division happens before multiplication in the listed instances.

[06] Inconsistent Use of NatSpec

NatSpec is a boon to all Solidity developers. Not only does it provide a structure for developers to document their code within the code itself, it encourages the practice of documenting code. When future developers read code documented with

NatSpec, they are able to increase their capacity to understand, upgrade, and fix code. Without code documented with NatSpec, that capacity is hindered.

The veRWA codebase does have a high level of documentation with NatSpec. However there are numerous instances of functions missing NatSpec.

Risks

1. Lack of understanding of code without adequate documentation.
2. Difficulty in understanding, upgrading, and fixing code without documentation.

Recommendation

1. Practice consistent use of NatSpec on all parts of the project codebase.
2. Include the need for NatSpec comments for code reviews to pass.

[07] Old version of OpenZeppelin Contracts used

Using old versions of 3rd-party libraries in the build process can introduce vulnerabilities to the protocol code.

- Latest is 4.9.3 (as of July 28, 2023), version used in project is 4.9.2

Risks

1. Older versions of OpenZeppelin contracts might not have fixes for known security vulnerabilities.
2. Older versions might contain features or functionality that have been deprecated in recent versions.
3. Newer versions often come with new features, optimizations, and improvements that are not available in older versions.

Recommendation

Review the latest version of the OpenZeppelin contracts and upgrade

[08] Error Messages don’t match intent

Meaningful error messages give better handles to test against. While building a test suite, edge cases can become numerous, and providing descriptive error messages allows not just for the project developers to write better tests, it helps developers of 3rd-party integrations and forks (other protocols, token projects, etc) to understand the intentions and reasoning of the parent project.

In the veRWA codebase, there are places where the error cases are materially different from the guards they are included with.

- From `VotingEscrow.sol` , line 573:

https://github.com/code-423n4/2023-08-verwa/blob/9a2e7be003bc1a77b3b87db31f3d5a1bcb48ed32/src/VotingEscrow.sol#L573

```
require(_blockNumber <= block.number, "Only past block number");
```

Risk

1. Project developers get the wrong understanding of why a case is erroneous.
2. Writing tests for specific error types becomes difficult because of a lack of unique errors provided by the executed code.
3. 3rd-party developers (forks, integrations) can have difficulties understanding developer intentions.

Recommendation

Use more descriptive handler names and error messages and limit reusing error messages for categorically-similar-yet-different errors.

[09] Function Names don’t match intent

`totalSupply` , `totalSupplyAt` , `supplyAt` in `VotingEscrow.sol` refer to voting power, not locked supply. This causes confusion to developers or auditors onboarding to the codebase.

[10] Dead Code

In many cases, dead code can create attack surfaces for malicious exploits. In the case for `VotingEscrow.sol` , there are instances of dead and unused code, though the risks are much lower.

Unused assignment

- From `_checkpoint()` in `VotingEscrow.sol`, line 206:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L206

```
    lastCheckpoint = iterativeTime; // <= this is never used
```

Unused Enums

Ex. VotingEscrow.sol, line 62 and 64

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L62

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L64

```
    enum LockAction {
                    ...
        INCREASE_TIME, // <= never used
        QUIT, // <= never used
                    ...
    }
```

Risk

While no attack vectors were identified from these instances of dead code, there is a definite increase in gas usage during execution and deployment.

Mitigation

Remove the instances of dead code from the codebase.

[11] TODOs included in code/comments

Ex. `LendingLedger.sol` , line 48 and `GaugeController.sol` line 59

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L48

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L59

```
    governance = _governance; // TODO: Maybe change to Oracle
```

[12] Reference to non-existent documentation

The references to IVotingEscrow are misleading because there is no IVotingEscrow anywhere in the codebase. This makes it difficult to trust comments in the codebase, let alone find the referenced documentation that may enlighten the developer or auditor.

Examples:

- https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L267
- https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L286
- https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L325
- https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L355

- https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L472https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L486

Audit Analysis

For this audit, 8 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **catellatech** received the top score from the judge.

The following wardens also submitted reports: rjs, MrPotatoMagic, RED-LOTUS-REACH, 0xSmartContract, 0x73696d616f, thekmj, *and* 0xbrett8571.

Description of `veRWA project`

veRWA is an incentivization mechanism designed for Real World Assets (RWA) on the Canto platform. It operates similarly to the veCRV system with its liquidity gauge. Users have the option to lock up CANTO tokens for a duration of five years in the `VotingEscrow` contract, in exchange for receiving veCANTO tokens.

Subsequently, they can participate in voting activities within the `GaugeController` for various credit markets that have been whitelisted by the governance. Users who provide liquidity in these credit markets can claim `CANTO` tokens (provided by the CANTO governance) from the `LendingLedger` contract, based on their proportional contribution.

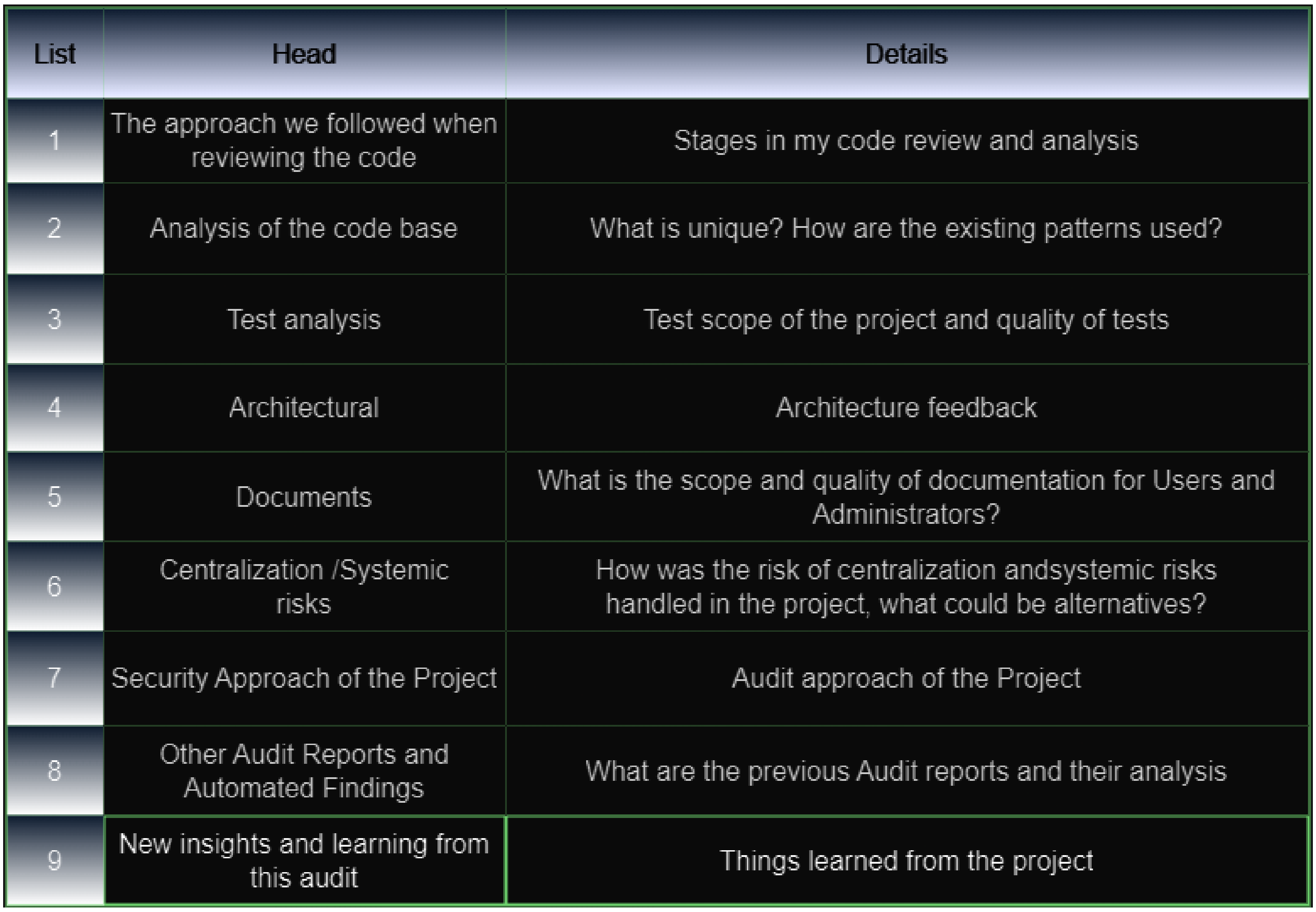
To illustrate, let's consider credit markets X, Y, and Z where Alice, Bob, and Charlie contribute liquidity. In the scenario of credit market X, assuming Alice contributes 60% of the liquidity, Bob contributes 30%, and Charlie contributes 10% at a specific time (epoch), and let's assume that credit market X receives 40% of all votes at that epoch. Consequently, the allocations would be:

- Alice: `40% * 60% = 24%` of all allocated CANTO tokens for this epoch.
- Bob: `40% * 30% = 12%` of all allocated CANTO tokens for this epoch.

- Charlie: `40% * 10% = 4%` of all allocated CANTO tokens for this epoch.

This mechanism aims to incentivize participation and liquidity provision in credit markets backed by real world assets on the Canto platform.

Summary



List	Head	Details
1	The approach we followed when reviewing the code	Stages in my code review and analysis
2	Analysis of the code base	What is unique? How are the existing patterns used?
3	Test analysis	Test scope of the project and quality of tests
4	Architectural	Architecture feedback
5	Documents	What is the scope and quality of documentation for Users and Administrators?
6	Centralization /Systemic risks	How was the risk of centralization andsystemic risks handled in the project, what could be alternatives?
7	Security Approach of the Project	Audit approach of the Project
8	Other Audit Reports and Automated Findings	What are the previous Audit reports and their analysis
9	New insights and learning from this audit	Things learned from the project

1- Approach we followed when reviewing the code

To begin, we assessed the code's scope, which guided our approach to reviewing and analyzing the project.

Scope

- **GaugeController.sol**:

  - The `GaugeController` contract is like a translated version of Curve's `GaugeController` written in Solidity. It simplifies the gauge types to just one type for veRWA. This leads to other changes in the code. The way gauges (lending markets) are approved is also different from the original design.

  The controller lets users vote on how much weight a gauge should have in distributing CANTO tokens during a specific period (epoch). It also allows checking the historical relative weights of all gauges.

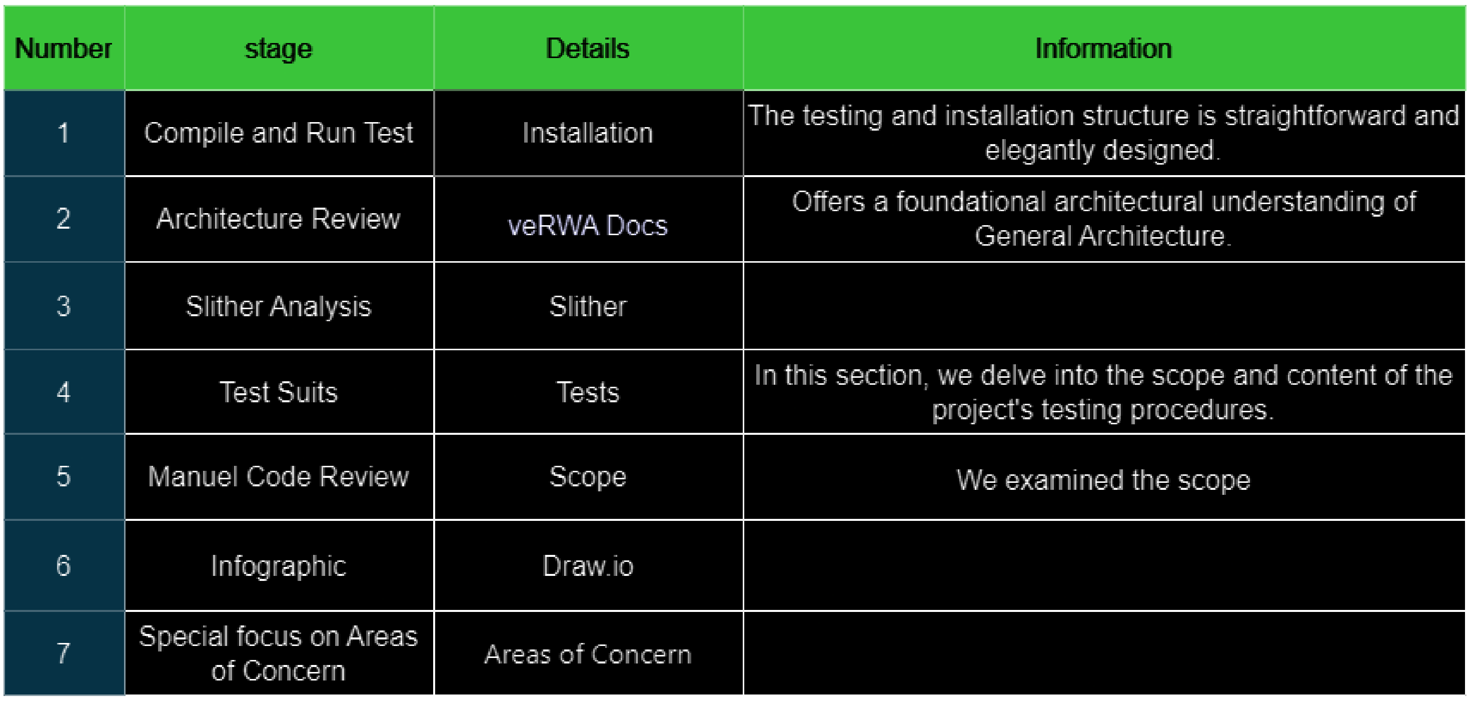
- **VotingEscrow.sol**:

  - The `VotingEscrow` system used here is based on an existing setup from FIAT DAO, which itself was built by modifying Curve's original design. In this version, a few changes were made, like allowing the use of native tokens instead of ERC20 tokens. Additionally, a fixed lock time of 5 years was added, which restarts every time an action is taken.

- **LendingLedger.sol**:

  - The lending ledger keeps track of how much cryptocurrency users have contributed to a specific market over time. To do this, authorized markets need to notify the ledger whenever a user deposits or withdraws funds. The Canto governance manages the rewards by sending CANTO tokens to the contract, controlling the amount allocated for each epoch.

With this understanding, we proceeded to scrutinize and audit the code through a series of structured steps:

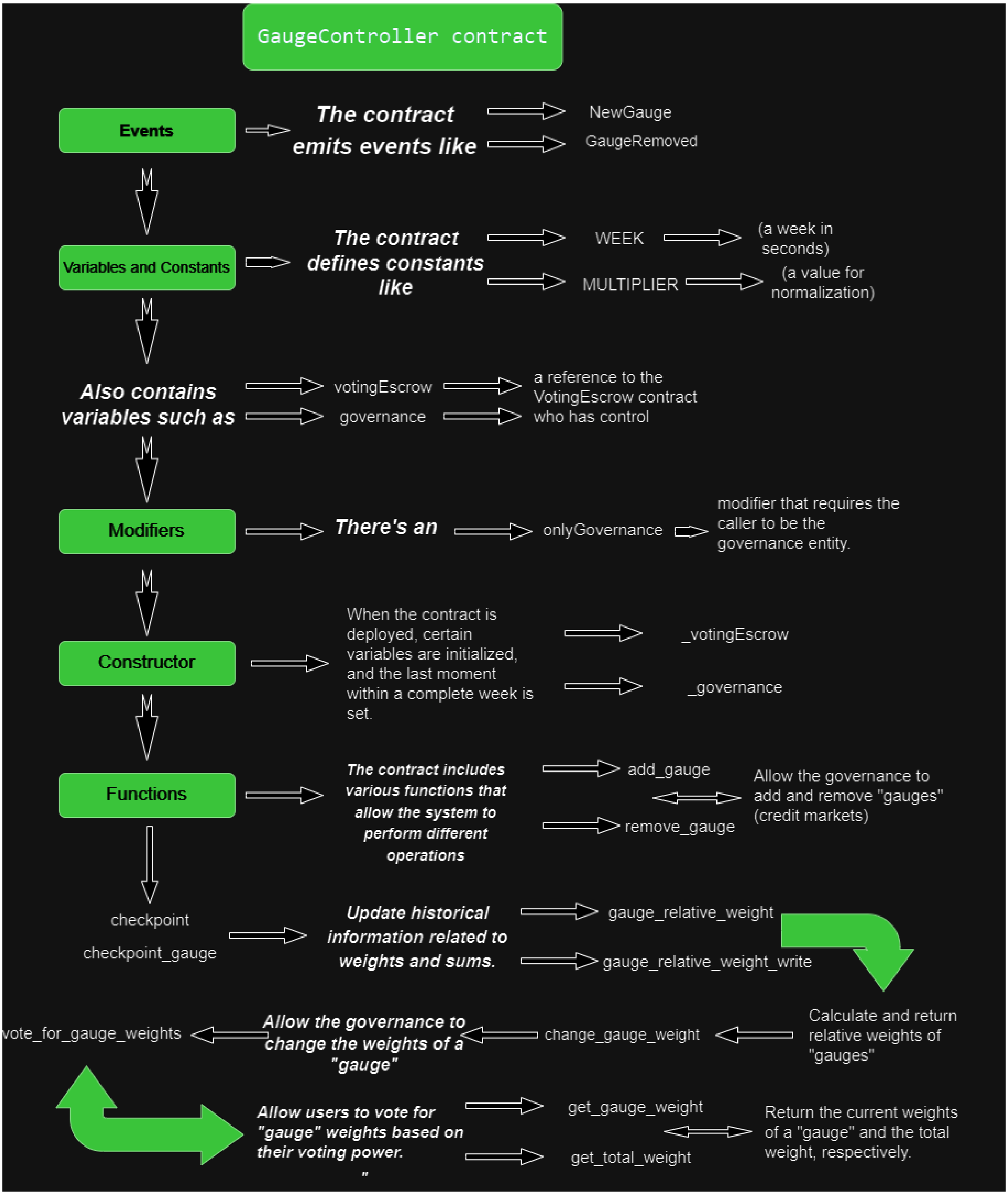


Number	stage	Details	Information
1	Compile and Run Test	Installation	The testing and installation structure is straightforward and elegantly designed.
2	Architecture Review	veRWA Docs	Offers a foundational architectural understanding of General Architecture.
3	Slither Analysis	Slither	
4	Test Suits	Tests	In this section, we delve into the scope and content of the project's testing procedures.
5	Manuel Code Review	Scope	We examined the scope
6	Infographic	Draw.io	
7	Special focus on Areas of Concern	Areas of Concern	

2- Analysis of the code base

The `GaugeController` **contract** is a fundamental component of the `veRWA project` is responsible for enabling users to vote on the distribution of CANTO tokens and managing information related to weights and changes in "gauges" (credit markets) within the `veRWA project`.

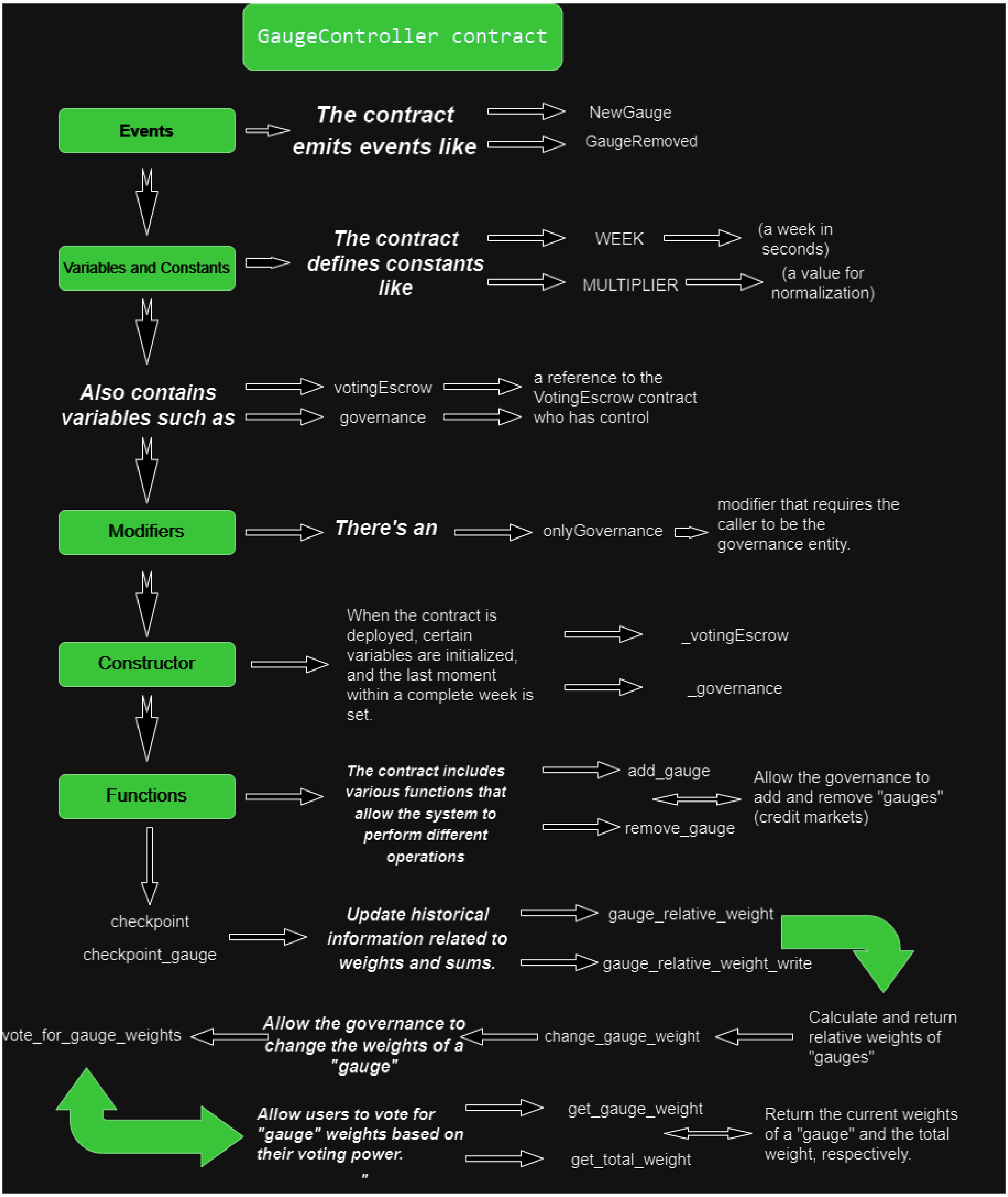
- Here's a breakdown of the key parts of this contract:



GaugeController contract
Events
The contract emits events like
NewGauge
GaugeRemoved
Variables and Constants
The contract defines constants like
WEEK
(a week in seconds)
MULTIPLIER
(a value for normalization)
Also contains variables such as
votingEscrow
a reference to the VotingEscrow contract
governance
who has control
Modifiers
There's an
onlyGovernance
modifier that requires the caller to be the governance entity.
Constructor
When the contract is deployed, certain variables are initialized, and the last moment within a complete week is set.
_votingEscrow
_governance
Functions
The contract includes various functions that allow the system to perform different operations
add_gauge
remove_gauge
Allow the governance to add and remove "gauges" (credit markets)
checkpoint
checkpoint_gauge
Update historical information related to weights and sums.
gauge_relative_weight
gauge_relative_weight_write
Calculate and return relative weights of "gauges"
change_gauge_weight
Allow the governance to change the weights of a "gauge"
vote_for_gauge_weights
Allow users to vote for "gauge" weights based on their voting power.
get_gauge_weight
get_total_weight
Return the current weights of a "gauge" and the total weight, respectively.

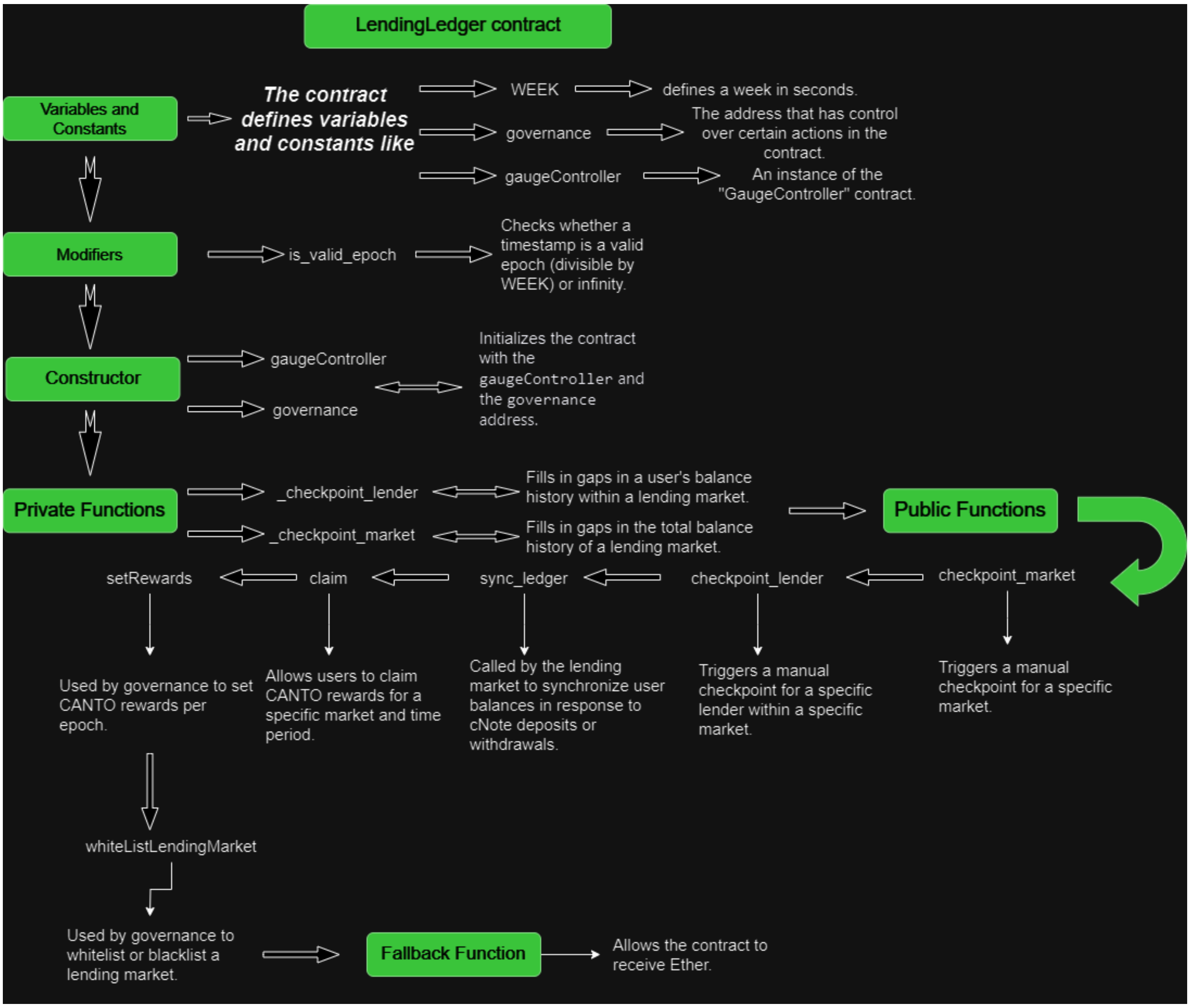
The `VotingEscrow` **contract** is responsible for managing users locking and voting power.

- I'll explain the key components and functionality of the contract in simpler terms:


GaugeController contract
Events
The contract emits events like
NewGauge
GaugeRemoved
Variables and Constants
The contract defines constants like
WEEK
(a week in seconds)
MULTIPLIER
(a value for normalization)
Also contains variables such as
votingEscrow
governance
a reference to the VotingEscrow contract
who has control
Modifiers
There's an
onlyGovernance
modifier that requires the caller to be the governance entity.
Constructor
When the contract is deployed, certain variables are initialized, and the last moment within a complete week is set.
_votingEscrow
_governance
Functions
The contract includes various functions that allow the system to perform different operations
add_gauge
remove_gauge
Allow the governance to add and remove "gauges" (credit markets)
checkpoint
checkpoint_gauge
Update historical information related to weights and sums.
gauge_relative_weight
gauge_relative_weight_write
Calculate and return relative weights of "gauges"
change_gauge_weight
Allow the governance to change the weights of a "gauge"
vote_for_gauge_weights
Allow users to vote for "gauge" weights based on their voting power.
get_gauge_weight
get_total_weight
Return the current weights of a "gauge" and the total weight, respectively.

The `LendingLedger` **contract** is responsible for tracking balances and rewards in lending markets. Users can synchronize their balances, claim rewards, and the governance can set rewards and control the whitelist of lending markets.

- Let's delve into its functionality and structure:


LendingLedger contract
Variables and Constants
The contract defines variables and constants like
WEEK
defines a week in seconds.
governance
The address that has control over certain actions in the contract.
gaugeController
An instance of the "GaugeController" contract.
Modifiers
is_valid_epoch
Checks whether a timestamp is a valid epoch (divisible by WEEK) or infinity.
Constructor
gaugeController
governance
Initializes the contract with the gaugeController and the governance address.
Private Functions
_checkpoint_lender
Fills in gaps in a user's balance history within a lending market.
_checkpoint_market
Fills in gaps in the total balance history of a lending market.
Public Functions
setRewards
claim
sync_ledger
checkpoint_lender
checkpoint_market
Used by governance to set CANTO rewards per epoch.
Allows users to claim CANTO rewards for a specific market and time period.
Called by the lending market to synchronize user balances in response to cNote deposits or withdrawals.
Triggers a manual checkpoint for a specific lender within a specific market.
Triggers a manual checkpoint for a specific market.
whiteListLendingMarket
Used by governance to whitelist or blacklist a lending market.
Fallback Function
Allows the contract to receive Ether.

3- Test analysis

The audit scope of the contracts to be audited is 95% and it should be aimed to be 100%.

```
What is the overall line coverage percentage provided by your te
```

How could they have done it better?

While the provided code seems to be functional, there are always areas for improvement to ensure better security, efficiency, and readability in both the contract and the protocol.

- Here are some potential areas for improvement:

  1. **Code Comments and Documentation:**

     - The code lacks detailed comments explaining the purpose and functionality of each function. Adding comprehensive comments can help other developers understand the codebase more easily.

  2. **Access Control:**

     - While the contract includes a onlyGovernance modifier for certain functions, it's essential to ensure that access control mechanisms are robust and properly implemented throughout the protocol. Consider using a more standardized access control library for clarity and security.

  3. **Error Handling and Assertions:**

     - The code uses require statements for error checking, which is good. However, more informative error messages could be added to provide better context about what went wrong during transactions.

  4. **Code Organization:**

     - The code could be organized into separate files or contracts for improved modularity and readability. This can make it easier to understand the different components of the protocol.

  5. **Security Audits:**

     - A thorough security audit by professionals can help identify vulnerabilities and potential attack vectors that might not be obvious. Security should always be a top priority.

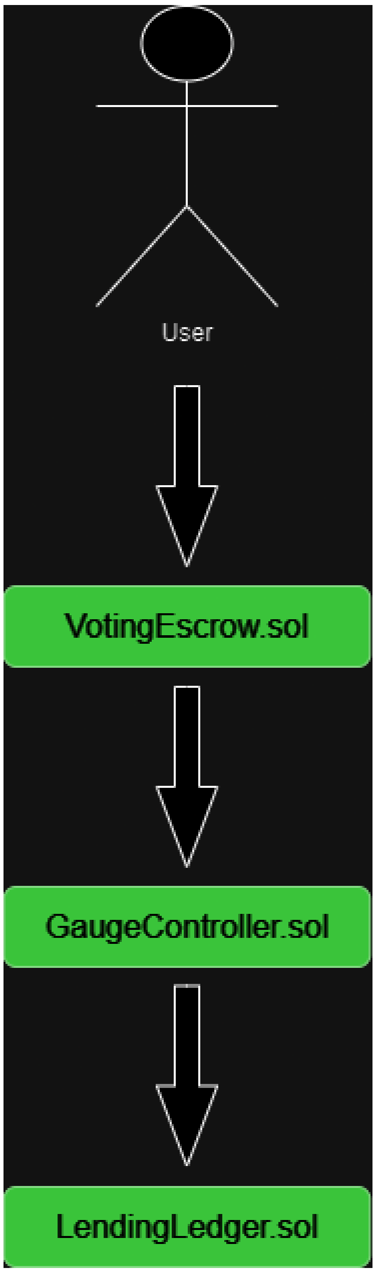
  6. **Event Logging:**

     - Adding well-defined event logs can help external services and users better understand the state changes and actions taken within the contract.

  7. **More Comprehensive Testing:**

- Extensive testing, including unit tests, integration tests, and possibly even external audits, can help ensure the robustness and correctness of the protocol.

4- Architectural



- **Users:** Users interact with the `veRWA project` through their actions, such as depositing tokens, voting on governance proposals, and participating in lending markets.
- `VotingEscrow.sol`:
  - This contract manages the voting and token locking system. Users can lock their tokens for a specified period and take part in community decision-making. The contract might be designed to encourage active participation and reward users for their voting actions.

- `GaugeController.sol` :

    - The `GaugeController` contract could manage the "gauges" that influence reward distribution. This contract likely defines how rewards are allocated among different markets or assets based on their relative weights.

- `LendingLedger.sol` :

    - This contract is the core of the lending and rewards functionality. Users can deposit tokens into lending markets, earn cTokens, and receive rewards in the form of `CANTO` tokens. The contract keeps track of user balances, total balances in each lending market, and other relevant metrics. It also allows users to claim rewards and enables governance to configure rewards.

This overview provides a general understanding of how the different components interact within the `veRWA project` .

5- Documents

- It would be helpful if the Documentation explained how the ecosystem works from a basic contract level so that it is easier to digest for developers, users and auditors looking to integrate into the `veRWA project` , it is recommended to add the architectural design to the documents as infographic
- I would also recommend adding quality Medium articles, it's a great way to provide an in-depth look at many of the topics in the project and is used by many blockchain projects.

6- Systemic & Centralization Risks

Here's an analysis of potential systemic and centralization risks in the provided contracts:

Systemic Risks:

- **Epoch-Based System:**

    - The protocol relies heavily on the concept of epochs (measured in weeks) for various functions like rewards, claims, and balances. If there's a flaw in how epochs are calculated or if the timing gets disrupted due to network issues, it could affect the accuracy of rewards and claims.

- **Reward Calculations:**
    - The calculation of rewards involves several variables, including market weights, balances, and reward amounts. If any of these variables are inaccurately calculated, it could lead to incorrect distribution of rewards, potentially causing dissatisfaction among users.
- **Lending Market Whitelist:**
    - The whitelist for lending markets is controlled by governance. If there’s centralized control over whitelisting and governance decisions, it could result in concentration of power and decision-making, which goes against the principles of decentralization.

Centralization Risks:

- **Governance Control:**
    - The governance address is hard-coded in the contract. If the governance becomes inactive or malicious, it could lead to a centralized decision-making process or even potential abuse of power.
- **Market Whitelisting:**
    - The ability for governance to whitelist or blacklist lending markets could lead to centralization if decisions are made without involving the broader community. Centralized control over market participation can hinder innovation and competition.
- **Reward Settings:**
    - Governance’s control over reward settings introduces centralization in determining how rewards are distributed. Mismanagement of rewards or a lack of transparency in setting rewards could lead to dissatisfaction among users.
- **Dependent Contracts:**

    - The protocol relies on external contracts like `VotingEscrow` and `GaugeController`. If these contracts are compromised or misconfigured, it could impact the overall functionality of the `veRWA project`.

- **Rewards Allocation:**

    - The distribution of rewards based on market weights controlled by `GaugeController` could be centralized if the controller's decisions are not well-distributed or transparently managed.

It's important to address these risks through careful auditing, community involvement, decentralization measures, and ongoing monitoring. The protocol should strive to minimize centralization, ensure transparency, and have contingency plans for potential disruptions or flaws in the system.

7- Security Approach of the Project

What the project should add in the understanding of Security?

- **Input Validation:**

    - Ensuring that inputs to functions are properly validated and sanitized to prevent unexpected behavior or vulnerabilities like integer overflows/underflows, division by zero, etc.

- **Access Control:**

    - Implementing proper access controls to restrict sensitive functions and data to authorized entities only. It seems that there is already an onlyGovernance modifier in the `LendingLedger.sol` contract for certain functions, which suggests some level of access control.

- **External Contract Interaction:**

    - Ensuring secure interactions with external contracts to prevent unexpected behavior or vulnerabilities. This involves validating return values, using established interfaces, and handling failures gracefully.

- **Audits and Testing:**

    - Conducting comprehensive security audits by reputable third-party auditors and performing extensive testing, including unit testing, integration testing, and scenario-based testing.
- **Code Simplicity and Clarity:**
    - Keeping the codebase simple and easy to understand, which can reduce the risk of introducing vulnerabilities due to complex logic.
- **Known Vulnerability Mitigation-:**
    - Staying updated with the latest developments in the blockchain and smart contract security space to mitigate known vulnerabilities.
- **Emergency Measures:**
    - Implementing mechanisms for pausing or upgrading contracts in case of vulnerabilities or emergencies.
- **Continuous Monitoring and Response:**
    - Implementing monitoring and response mechanisms to detect anomalies or attacks and respond to them in a timely manner.

8- Other Audit Reports and Automated Findings

Automated Findings: https://github.com/code-423n4/2023-08-verwa/blob/main/bot-report.md

9- New insights and learning from this audit

The insights and learnings that could be gained from analyzing the provided smart contracts from the `veRWA project`.

Complexity and Design Patterns: - Analyzing these contracts might offer insights into how complex DeFi protocols are designed and implemented. Understanding the interplay of different contracts, data structures, and design patterns could deepen your understanding of decentralized applications.

Governance and Voting: - By studying the `VotingEscrow` contract, you could learn about how decentralized governance and voting mechanisms are implemented. This could be valuable for understanding how community-driven decisions are made in various projects.

Tokenization and Rewards:

- The use of ERC-20 tokens (like CANTO) for rewards and incentives demonstrates the tokenization of value within DeFi ecosystems. Learning about how tokens are distributed as rewards can provide insights into incentivizing users' participation.

Conclusion

In general, the `veRWA project` exhibits an interesting and well-developed architecture we believe the team has done a good job regarding the code, but the identified risks need to be addressed, and measures should be implemented to protect the protocol from potential malicious use cases. Additionally, it is recommended to improve the documentation and comments in the code to enhance understanding and collaboration among developers. It is also highly recommended that the team continues to invest in security measures such as mitigation reviews, audits, and bug bounty programs to maintain the security and reliability of the project.

Time Spent ⏱️

A total of 2 days were spent to cover this audit, broken down into the following:

1. 1st Day: Trying to understand the protocol flows and implementation
2. 2nd Day: We focused on conducting the most comprehensive analysis possible, adding handcrafted diagrams based on the contracts and information provided by the protocol.

Time spent: 24 hours

OpenCoreCH (veRWA) acknowledged and commented:

> Great report! I especially like the diagram.

alcueca (Judge) commented:

> The diagrams in this report are representative of the effort put into it. Additionally, sections 5 (Documents), 6 (Risks) and 7 (Security Approach) complete the report further by providing additional insight.

> While the application description in #452 is more thorough and detailed, and should be reviewed by those exploring the codebase, this report offers more value for the sponsor, and is therefore selected for the report.

Disclosures