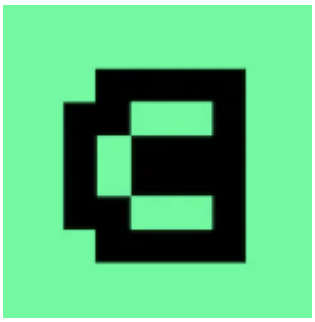
Canto veRWA



Scope

The code under review can be found within the C4 veRWA repository.

Summary

Findings

ID	Issue	Severity
H-01	Users will not be able to withdraw CANTO if their lock expires under existing delegation	High
H-02	User will not be able to withdraw CANTO if they delegate their power to another address	High
L-01	Require check incorrectly implemented	Low
L-02	Loss of precision due to division occurring before multiplication	Low
L-03	Slope can round down to zero if numerator is smaller than denominator	Low
L-04	LOCKTIME does not consider for 2 or more extra days introduced due to leap years	Low
L-05	Consider skipping the epoch for which rewards are already set	Low
N-01	Public variables only used within contract can be changed to private visibility	Non-Critical
N-02	Variables that are unchanging should be marked constant or immutable if assigned in constructor	Non-Critical
N-03	Missing event emission for critical storage changes in functions	Non-Critical

ID	Issue	Severity
N-04	Remove redundant code and comments to improve code readability and maintainability	Non-Critical
N-05	Consider using delete instead of assigning address(0) to clear values	Non-Critical
N-06	Require statements should provide an error message in case condition fails	Non-Critical
N-07	Modify code to improve code readability and maintainability	Non-Critical
N-08	Return values not handled in functions	Non-Critical
N-09	Duplicated require/if statements should be refactored	Non-Critical

Analysis Report

- Comments for the judge to contextualize findings
- Approach taken in evaluating the codebase
- Architecture recommendations
- Codebase quality analysis
- Centralization risks
- Mechanism review
- Time spent

Findings

[H-01] Users will not be able to withdraw CANTO if their lock expires under existing delegation

Impact

If the user has delegated their power to another address and their lock expires, the user will not be able to withdraw their CANTO.

Proof of Concept

There are two steps involved in this:

Address A = User's address (I'll be using the terms "User" and "address A" interchangeably)
Address B = The address User delegates his power to

1. User delegates power to B 2. User un-delegates power from address B and delegates back to his address A

Let's understand the first step in a detailed manner:

1. User calls the delegate() function with parameter _addr to delegate his power to address B. **(Note: We pass the checks on Line 359,360 since user has a valid lock and is delegating to a different address)**

```
File: src/VotingEscrow.sol
356:     function delegate(address _addr) external nonReentrant {
357:         LockedBalance memory locked_ = locked[msg.sender];
358:         // Validate inputs
359:         require(locked_.amount > 0, "No lock");
360:         require(locked_.delegatee != _addr, "Already delegated");
361:         // Update locks
362:         int128 value = locked_.amount;
363:         address delegatee = locked_.delegatee;
364:         LockedBalance memory fromLocked;
365:         LockedBalance memory toLocked;
366:         locked_.delegatee = _addr;
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:         } else if (_addr == msg.sender) {
372:             // Undelegate
373:             fromLocked = locked[delegatee];
374:             toLocked = locked_;
375:         } else {
376:             // Re-delegate
377:             fromLocked = locked[delegatee];
378:             toLocked = locked[_addr];
379:             // Update owner lock if not involved in delegation
380:             locked[msg.sender] = locked_;
381:         }
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
385:         _delegate(delegatee, fromLocked, value,
LockAction.UNDELEGATE);
386:         _delegate(_addr, toLocked, value, LockAction.DELEGATE);
387:     }
```

2. On Line 363 and 366, we store the current locked.delegatee (i.e. the user or msg.sender) in variable **delegatee** and update the locked.delegatee to the new address B.

```
File: src/VotingEscrow.sol
363:         address delegatee = locked_.delegatee; //@audit delegatee =
msg.sender
366:         locked_.delegatee = _addr; //@audit sets _addr to be new
delegatee
```

3. Since the **delegatee** is the msg.sender (i.e. the user) , we enter the first if block. This stores the LockedBalance of both the user and address B in the **fromLocked** and **toLocked** variables respectively. The only update made is in the fromLocked since we change the delegatee to the new address B on Line 366.

```
File: src/VotingEscrow.sol
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:         }
```

4. Next up, we come across a bunch of require checks that demand the new delegatee (i.e. address B) to have a valid lock which has not expired and has a longer end time. But the most important check here (we'll see why this is important ahead) is the one on Line 383 which demands the new delegatee (address B) to have a lock that has not expired.

```
File: src/VotingEscrow.sol
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
```

5. We pass the above checks and next call the _delegate() function to un-delegate the voting power from the user (Line 385) and delegate that voting power to address B (Line 386).

```
File: src/VotingEscrow.sol
385:         _delegate(delegatee, fromLocked, value,
LockAction.UNDELEGATE);
386:         _delegate(_addr, toLocked, value, LockAction.DELEGATE);
```

6. The _delegate() function updates the voting power correctly for both addresses and calls the _checkPoint() function to record history appropriately. **(Note: I'm not explaining the functionality here since the problem lies in the delegate() function. Let's assume this works as intended.)**

```
File: src/VotingEscrow.sol
390:     function _delegate(
391:         address addr,
392:         LockedBalance memory _locked,
393:         int128 value,
394:         LockAction action
395:     ) internal {
396:         LockedBalance memory newLocked = _copyLock(_locked);
397:         if (action == LockAction.DELEGATE) {
```

```
398:             newLocked.delegated += value;
399:             emit Deposit(addr, uint256(int256(value)), newLocked.end,
action, block.timestamp);
400:         } else {
401:             newLocked.delegated -= value;
402:             emit Withdraw(addr, uint256(int256(value)), action,
block.timestamp);
403:         }
404:         locked[addr] = newLocked;
405:         if (newLocked.amount > 0) {
406:             // Only if lock (from lock) hasn't been withdrawn/quitted
407:             _checkpoint(addr, _locked, newLocked);
408:         }
409:     }
```

This marks the end of the first step where the user delegated his power to address B sucessfully.

Let's understand Step 2 now where the user's **lock has expired** and he tries to un-delegate power from address B and delegate back to his address A:

1. User calls the delegate() function with his address A as parameter. **(Note: We pass the checks on Line 359,360 since user has a valid lock and is delegating to a different address i.e. his own address A)**

```
File: src/VotingEscrow.sol
356:     function delegate(address _addr) external nonReentrant {
357:         LockedBalance memory locked_ = locked[msg.sender];
358:         // Validate inputs
359:         require(locked_.amount > 0, "No lock");
360:         require(locked_.delegatee != _addr, "Already delegated");
361:         // Update locks
362:         int128 value = locked_.amount;
363:         address delegatee = locked_.delegatee;
364:         LockedBalance memory fromLocked;
365:         LockedBalance memory toLocked;
366:         locked_.delegatee = _addr;
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:         } else if (_addr == msg.sender) {
372:             // Undelegate
373:             fromLocked = locked[delegatee];
374:             toLocked = locked_;
375:         } else {
376:             // Re-delegate
377:             fromLocked = locked[delegatee];
378:             toLocked = locked[_addr];
379:             // Update owner lock if not involved in delegation
380:             locked[msg.sender] = locked_;
381:         }
```

```
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
385:         _delegate(delegatee, fromLocked, value,
LockAction.UNDELEGATE);
386:         _delegate(_addr, toLocked, value, LockAction.DELEGATE);
387:     }
```

2. On Line 363 and 366, we store the current locked.delegatee (i.e. address B) in variable **delegatee** and update the locked.delegatee to the user's address A.

```
File: src/VotingEscrow.sol
363:         address delegatee = locked_.delegatee; //@audit delegatee =
msg.sender
366:         locked_.delegatee = _addr; //@audit sets _addr to be new
delegatee
```

3. Since the **_addr** is address A , we enter the second else if block. This stores the LockedBalance of both address B and the user's address A in the **fromLocked** and **toLocked** variables respectively. The only update made is in the toLocked since we change the delegatee to the user's address A on Line 366.

```
File: src/VotingEscrow.sol
371:         else if (_addr == msg.sender) {
372:             // Undelegate
373:             fromLocked = locked[delegatee];
374:             toLocked = locked_;
375:         }
```

4. Here is where the trick now comes into play. The first and third require checks demand the user to have a valid lock with a longer end time (Let's assume we pass these checks). But the second check on Line 383 demands the user's end time (i.e. toLocked.end) to be greater than block.timestamp (i.e. to not have expired). We fail this condition since the user's lock has expired in our case. Due to this, we revert and the user is not able to change the delegation from address B back to his address A.

```
File: src/VotingEscrow.sol
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
```

5. Since the delegation cannot be switched back to address A due to the above problem, this prevents the user from withdrawing CANTO tokens since in the withdraw() function below, the require check on Line 331 demands that the delegatee should be the msg.sender (i.e. the user).

```
File: src/VotingEscrow.sol
326:     function withdraw() external nonReentrant {
327:         LockedBalance memory locked_ = locked[msg.sender];
328:         // Validate inputs
329:         require(locked_.amount > 0, "No lock");
330:         require(locked_.end <= block.timestamp, "Lock not expired");
331:         require(locked_.delegatee == msg.sender, "Lock delegated");
332:         // Update lock
333:         uint256 amountToSend = uint256(uint128(locked_.amount));
334:         LockedBalance memory newLocked = _copyLock(locked_);
335:         newLocked.amount = 0;
336:         newLocked.end = 0;
337:         newLocked.delegated -= int128(int256(amountToSend));
338:         newLocked.delegatee = address(0);
339:         locked[msg.sender] = newLocked;
340:         newLocked.delegated = 0;
341:         // oldLocked can have either expired <= timestamp or zero end
342:         // currentLock has only 0 end
343:         // Both can have >= 0 amount
344:         _checkpoint(msg.sender, locked_, newLocked);
345:         // Send back deposited tokens
346:         (bool success, ) = msg.sender.call{value: amountToSend}("");
347:         require(success, "Failed to send CANTO");
348:         emit Withdraw(msg.sender, amountToSend, LockAction.WITHDRAW,
block.timestamp);
349:     }
```

These two steps show us how delegation could be changed from address A to address B but not back from address B to address A due to the user's lock expired problem. Additionally, the delegation problem showed us how withdrawal of CANTO tokens is disallowed, thereby locking the user's CANTO.

Things to note: 1. A new lock cannot be created through createLock() as well since the previous lock has not been withdrawn. 2. I believe this issue is not the user's mistake since the user knows that they can only withdraw their CANTO after their lock has expired. Therefore, it is natural for them to undelegate any existing delegation after their lock has expired and then followingly withdraw CANTO. 3. This issue is different from my previous issue number #58 submitted as the root cause in this is the user's lock expiration while in issue number #58 it is the user not having a longer end time. Even if we resolve the issue in #58, the issue still persists in this case.)

Here is a Coded POC which proves that delegation back to user's address A is not possible after lock has expired:

1. Add this test at the end of the VotingEscrow.t.sol file
2. Run the command: `forge test --match-test testDelegationBackToMsgSenderSuccessAfterLockExpired -vvvv`

```
File: src/VotingEscrow.t.sol
358:     function testDelegationBackToMsgSenderSuccessAfterLockExpired()
public {
359:         vm.prank(user1);
360:         ve.createLock{value: LOCK_AMT}(LOCK_AMT); //user1 creates a
lock
361:         vm.prank(user2);
362:         ve.createLock{value: LOCK_AMT}(LOCK_AMT); //user2 creates a
lock
363:         vm.prank(user1);
364:         ve.delegate(user2);//user1 delegates power to user2
365:         vm.warp(block.timestamp + 1826 days); //warp 1826 days ahead
since after 1825 days is when user's lock expires (i.e. this indicates
user's lock has now expired)
366:         vm.prank(user1);
367:         ve.delegate(user1);//user1 tries to undelegate user2 and
delegate back to himself (this should succeed but it fails since user's
lock has expired)
368:     }
```

Tools Used

Manual Review

Recommended Mitigation Steps

We know the require check causing the problem is important for other cases. Therefore, the best solution would be to move the require check in the if and else blocks but not the else if block like this (updates made on Line 371,382):

```
File: src/VotingEscrow.sol
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:             require(toLocked.end > block.timestamp, "Delegatee lock
expired");//@audit check added here
372:         } else if (_addr == msg.sender) {
373:             // Undelegate
374:             fromLocked = locked[delegatee];
375:             toLocked = locked_;
376:         } else {
377:             // Re-delegate
378:             fromLocked = locked[delegatee];
379:             toLocked = locked[_addr];
380:             // Update owner lock if not involved in delegation
381:             locked[msg.sender] = locked_;
382:             require(toLocked.end > block.timestamp, "Delegatee lock
expired");//@audit check added here
383:         }
```

Note: Do not forget to remove the require check on Line 383 when making these changes.

[H-02] User will not be able to withdraw CANTO if they delegate their power to another address

Impact

User will not be able to withdraw their CANTO tokens if they delegate their power to another address through the delegate() function.

Proof of Concept

There are two steps involved in this:

Address A = User's address (I'll be using the terms "User" and "address A" interchangeably)
Address B = The address User delegates his power to

1. User delegates power to B 2. User un-delegates power from address B and delegates back to his address A

Let's understand the first step in a detailed manner:

1. User calls the delegate() function with parameter _addr to delegate his power to address B. **(Note: We pass the checks on Line 359,360 since user has a valid lock and is delegating to a different address)**

```
File: src/VotingEscrow.sol
356:     function delegate(address _addr) external nonReentrant {
357:         LockedBalance memory locked_ = locked[msg.sender];
358:         // Validate inputs
359:         require(locked_.amount > 0, "No lock");
360:         require(locked_.delegatee != _addr, "Already delegated");
361:         // Update locks
362:         int128 value = locked_.amount;
363:         address delegatee = locked_.delegatee;
364:         LockedBalance memory fromLocked;
365:         LockedBalance memory toLocked;
366:         locked_.delegatee = _addr;
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:         } else if (_addr == msg.sender) {
372:             // Undelegate
373:             fromLocked = locked[delegatee];
374:             toLocked = locked_;
375:         } else {
376:             // Re-delegate
377:             fromLocked = locked[delegatee];
378:             toLocked = locked[_addr];
```

```
379:             // Update owner lock if not involved in delegation
380:             locked[msg.sender] = locked_;
381:         }
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
385:         _delegate(delegatee, fromLocked, value,
LockAction.UNDELEGATE);
386:         _delegate(_addr, toLocked, value, LockAction.DELEGATE);
387:     }
```

2. On Line 363 and 366, we store the current locked.delegatee (i.e. the user or msg.sender) in variable **delegatee** and update the locked.delegatee to the new address B.

```
File: src/VotingEscrow.sol
363:         address delegatee = locked_.delegatee; //@audit delegatee =
msg.sender
366:         locked_.delegatee = _addr; //@audit sets _addr to be new
delegatee
```

3. Since the **delegatee** is the msg.sender (i.e. the user) , we enter the first if block. This stores the LockedBalance of both the user and address B in the **fromLocked** and **toLocked** variables respectively. The only update made is in the fromLocked since we change the delegatee to the new address B on Line 366.

```
File: src/VotingEscrow.sol
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:         }
```

4. Next up, we come across a bunch of require checks that demand the new delegatee (i.e. address B) to have a valid lock which has not expired. But the most important check here (we'll see why this is important ahead) is the one on Line 384 which demands the new delegatee (address B) to have a longer lock than the user's lock.

```
File: src/VotingEscrow.sol
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
```

5. We pass the above checks and next call the _delegate() function to un-delegate the voting power from the user (Line 385) and delegate that voting power to address B (Line 386).

```
File: src/VotingEscrow.sol
385:         _delegate(delegatee, fromLocked, value,
LockAction.UNDELEGATE);
386:         _delegate(_addr, toLocked, value, LockAction.DELEGATE);
```

6. The _delegate() function updates the voting power correctly for both addresses and calls the _checkPoint() function to record history appropriately. **(Note: I'm not explaining the functionality here since the problem lies in the delegate() function. Let's assume this works as intended.)**

```
File: src/VotingEscrow.sol
390:     function _delegate(
391:         address addr,
392:         LockedBalance memory _locked,
393:         int128 value,
394:         LockAction action
395:     ) internal {
396:         LockedBalance memory newLocked = _copyLock(_locked);
397:         if (action == LockAction.DELEGATE) {
398:             newLocked.delegated += value;
399:             emit Deposit(addr, uint256(int256(value)), newLocked.end,
action, block.timestamp);
400:         } else {
401:             newLocked.delegated -= value;
402:             emit Withdraw(addr, uint256(int256(value)), action,
block.timestamp);
403:         }
404:         locked[addr] = newLocked;
405:         if (newLocked.amount > 0) {
406:             // Only if lock (from lock) hasn't been withdrawn/quitted
407:             _checkpoint(addr, _locked, newLocked);
408:         }
409:     }
```

This marks the end of the first step where the user delegated his power to address B sucessfully.

Let's understand Step 2 now where the user un-delegates power from address B and delegates back to his address A:

1. User calls the delegate() function with his address A as parameter. **(Note: We pass the checks on Line 359,360 since user has a valid lock and is delegating to a different address i.e. his own address A)**

```
File: src/VotingEscrow.sol
356:     function delegate(address _addr) external nonReentrant {
```

```
357:         LockedBalance memory locked_ = locked[msg.sender];
358:         // Validate inputs
359:         require(locked_.amount > 0, "No lock");
360:         require(locked_.delegatee != _addr, "Already delegated");
361:         // Update locks
362:         int128 value = locked_.amount;
363:         address delegatee = locked_.delegatee;
364:         LockedBalance memory fromLocked;
365:         LockedBalance memory toLocked;
366:         locked_.delegatee = _addr;
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:         } else if (_addr == msg.sender) {
372:             // Undelegate
373:             fromLocked = locked[delegatee];
374:             toLocked = locked_;
375:         } else {
376:             // Re-delegate
377:             fromLocked = locked[delegatee];
378:             toLocked = locked[_addr];
379:             // Update owner lock if not involved in delegation
380:             locked[msg.sender] = locked_;
381:         }
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
385:         _delegate(delegatee, fromLocked, value,
LockAction.UNDELEGATE);
386:         _delegate(_addr, toLocked, value, LockAction.DELEGATE);
387:     }
```

2. On Line 363 and 366, we store the current locked.delegatee (i.e. address B) in variable **delegatee** and update the locked.delegatee to the user's address A.

```
File: src/VotingEscrow.sol
363:         address delegatee = locked_.delegatee; //@audit delegatee =
msg.sender
366:         locked_.delegatee = _addr; //@audit sets _addr to be new
delegatee
```

3. Since the **delegatee** is address B , we enter the second else if block. This stores the LockedBalance of both address B and the user's address A in the **fromLocked** and **toLocked** variables respectively. The only update made is in the toLocked since we change the delegatee to the user's address A on Line 366.

```
File: src/VotingEscrow.sol
371:         else if (_addr == msg.sender) {
372:             // Undelegate
373:             fromLocked = locked[delegatee];
374:             toLocked = locked_;
375:         }
```

4. Here is where the trick now comes into play. The first two require checks demand the user to have a valid lock which has not expired (Let's assume we pass these checks). But the third check on Line 384 demands the user's end time (i.e. toLocked.end) to be greater than address B's end time (i.e. fromLocked.end). We fail this condition since in the previous delegation from address A to address B, we required address B's end time to be greater than the user's end time). Due to this, we revert and the user is not able to change the delegation from address B back to his address A.

```
File: src/VotingEscrow.sol
382:         require(toLocked.amount > 0, "Delegatee has no lock");
383:         require(toLocked.end > block.timestamp, "Delegatee lock
expired");
384:         require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");
```

5. Since the delegation cannot be switched back to address A due to the above problem, this prevents the user from withdrawing CANTO tokens since in the withdraw() function below, the require check on Line 331 demands that the delegatee should be the msg.sender (i.e. the user).

```
File: src/VotingEscrow.sol
326:     function withdraw() external nonReentrant {
327:         LockedBalance memory locked_ = locked[msg.sender];
328:         // Validate inputs
329:         require(locked_.amount > 0, "No lock");
330:         require(locked_.end <= block.timestamp, "Lock not expired");
331:         require(locked_.delegatee == msg.sender, "Lock delegated");
332:         // Update lock
333:         uint256 amountToSend = uint256(uint128(locked_.amount));
334:         LockedBalance memory newLocked = _copyLock(locked_);
335:         newLocked.amount = 0;
336:         newLocked.end = 0;
337:         newLocked.delegated -= int128(int256(amountToSend));
338:         newLocked.delegatee = address(0);
339:         locked[msg.sender] = newLocked;
340:         newLocked.delegated = 0;
341:         // oldLocked can have either expired <= timestamp or zero end
342:         // currentLock has only 0 end
343:         // Both can have >= 0 amount
344:         _checkpoint(msg.sender, locked_, newLocked);
345:         // Send back deposited tokens
346:         (bool success, ) = msg.sender.call{value: amountToSend}("");
```

```
347:         require(success, "Failed to send CANTO");
348:         emit Withdraw(msg.sender, amountToSend, LockAction.WITHDRAW,
block.timestamp);
349:     }
```

These two steps show us how delegation could be changed from address A to address B but not back from address B to address A due to the longer end time problem. Additionally, the delegation problem showed us how withdrawal of CANTO tokens is disallowed, thereby locking the user's CANTO.

Tools Used

Manual Review

Recommended Mitigation Steps

We know the require check causing the problem is important for other cases. Therefore, the best solution would be to move the require check in the if and else blocks but not the else if block like this **(updates made on Line 371,382)**:

```
File: src/VotingEscrow.sol
367:         if (delegatee == msg.sender) {
368:             // Delegate
369:             fromLocked = locked_;
370:             toLocked = locked[_addr];
371:             require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");//@audit check added here
372:         } else if (_addr == msg.sender) {
373:             // Undelegate
374:             fromLocked = locked[delegatee];
375:             toLocked = locked_;
376:         } else {
377:             // Re-delegate
378:             fromLocked = locked[delegatee];
379:             toLocked = locked[_addr];
380:             // Update owner lock if not involved in delegation
381:             locked[msg.sender] = locked_;
382:             require(toLocked.end >= fromLocked.end, "Only delegate to
longer lock");//@audit check added here
383:         }
```

Note: Do not forget to remove the require check on Line 384 when making these changes.

[L-01] Require check incorrectly implemented

There is 1 instance of this issue:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L294

The check below is implemented as follows with the > operator:

```
File: src/VotingEscrow.sol
294: require(locked_.end > block.timestamp, "Lock expired");
```

But the `@dev` tag on Line 287 mentions it should use `>=` operator:

```
File: src/VotingEscrow.sol
287: // @dev A lock is active until both lock.amount==0 and
lock.end<=block.timestamp
```

Solution:

```
File: src/VotingEscrow.sol
294: require(locked_.end >= block.timestamp, "Lock expired");
```

[L-02] Loss of precision due to division occurring before multiplication

(Note: This is different from the [L-08] bot finding

Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division might truncate. See here

Most of the instances below are timing-related, thus it is important to ensure that any operations that determine time are as precise as possible so that any reads and writes from/to storage are synced and correct. There are 9 instances of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L425

```
File: src/VotingEscrow.sol
425: return (_t / WEEK) * WEEK;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L60C10-L60C10

```
File: src/GaugeController.sol
60: uint256 last_epoch = (block.timestamp / WEEK) * WEEK;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L153

```
```

```
File: src/GaugeController.sol
153: uint256 t = (_time / WEEK) * WEEK;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L191

```
File: src/GaugeController.sol
191: uint256 next_time = ((block.timestamp + WEEK) / WEEK) * WEEK;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L224

```
File: src/GaugeController.sol
224: uint256 next_time = ((block.timestamp + WEEK) / WEEK) * WEEK;
```

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

```
File: src/LendingLedger.sol
60: uint256 currEpoch = (block.timestamp / WEEK) * WEEK;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L84

```
File: src/LendingLedger.sol
84: uint256 currEpoch = (block.timestamp / WEEK) * WEEK;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L134

```
File: src/LendingLedger.sol
134: uint256 currEpoch = (block.timestamp / WEEK) * WEEK;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L162

```
File: src/LendingLedger.sol
162: uint256 currEpoch = (block.timestamp / WEEK) * WEEK;
```

[L-03] Slope can round down to zero if numerator is smaller than denominator

(Note: The instance below is not included in the [L-08] bot finding)

Solidity doesn't support fractions, so divisions by large numbers could result in the quotient being zero. To avoid this, it's recommended to require a minimum numerator amount to ensure that it is always greater than the denominator.

There is 1 instance of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L231

If `slope * _user_weight` is less than 10_000, it can lead to rounding to zero and incorrect evaluations in the statements following it.

```
File: src/GaugeController.sol
231:  slope: (slope * _user_weight) / 10_000,
```

[L-04] LOCKTIME does not consider for 2 or more extra days introduced due to leap years

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L32

The current LOCKTIME considers that all 5 years are 365 days long. We could let this by if there was only leap year that the lock comes across during its lifespan of 5 years. But that is not the case here.

Let's take three examples:

1. Suppose a lock is created in 2024, the end time for it would be 2029. Since 2024 to 2029 includes two leap years (2024 and 2028), this adds 2 extra days. But since the locktime considers each year to be 365 days long, we skip these 2 extra days. (Note: It is highly likely such an issue occurs as 2024 is next year and with the launch of this fresh codebase, there are high chances of locks being created in 2024).

2. Suppose a lock is created in 2023, the end time for it would be 2028. Since 2023 to 2028 includes one leap year (2024, the lock ends at start of 2028). this adds 1 extra day. We can let this by but if the lock is extended by increaseAmount() function, the introduction of a second leap year (2028) is possible. In that case, two extra days are not added to the lifespan of the lock.

3. If any lock created is extended with increaseAmount() function, that will introduce an additional leap year (i.e. an extra day) which is not accounted for in the lifespan of the contract. This adds up if the user keeps on extending their lock.

Solution: If you go to see technically, each year is 365 days and 6 hours long. That is why we have a leap year every 4 years (6 * 4 = 24 hours = 1 day) since the 6 hours are accounted in the leap year. The solution to this problem would be to change the LOCKTIME from 1825 days to 1828 days. This is because 1825/5 = 365 days but 1828/5 = 365.6 (i.e. 1828 days accounts for the 6 hours in each year, thus now making each

year 365 days and 6 hours long rather than 365 days only). Note that there is no rounding error occurring here since when creating a lock, we are just adding the days to block.timestamp, which is then rounded down to weeks.

```
File: src/VotingEscrow.sol
32: uint256 public constant LOCKTIME = 1825 days; //@audit make this 1828
days
```

[L-05] Consider skipping the epoch for which rewards are already set

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L195

Here is the discussion between the sponsor and I to gain some context before reading my solution:

Warden:

```
Hi, over here do you think it would make sense to just "continue" the for
loop rather than revert if rewards are already set? That way it might be
easy for the governance to set the rewards as well I think.
https://github.com/code-423n4/2023-08-
verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#
L195
```

Sponsor:

```
I revert there on purpose because if there is a governance proposal to set
rewards for epochs X to X + Y, I think it should either set it for all
epochs or fail (and require a new one). Only setting it for a few could be
a bit confusing imo, but would of course also work
```

Warden:

```
hmm what I was thinking is if we set rewards for epochs X to X + Y, there
might be an epoch X + 1 which might have rewards already set and there is
no way to set it's ri.set to false. Therefore the governance might need to
set the rewards as X to X and X + 2 to X + Y to go around X + 1.  I think
it is highly unlikely something like this would occur but the case is
still valid I believe. Lmk what you think about this.
```

Sponsor:

```
Yes exactly, in such a scenario I would prefer two separate proposals with
[X, X] and [X + 2, X + Y] because I think the danger with one proposal for
[X, X + Y] is that voters expect that the rewards will be set for the
whole range
```

Warden:

```
hmm your right that makes sense from the voters perspective for ease of
understandability and I believe is the way to go. Just had an additional
point that if there is more than 1 epoch for which rewards are set then I
believe it might be time consuming and a bit heavy work-wise on the
voter's side. Again the likelihood for this is low.
```

Solution:

If ri.set is false it can lead to reversion of all previous storage updates. This can prevent the governance from setting rewards per epoch in the range fromEpoch to toEpoch. Although this reversion is the desired behaviour (i.e. what the sponsor expects - discussed with sponsor) and can be solved with two separate proposals to go around the epoch (rewards for which are already set), I think it might be time consuming and a bit heavy work-wise on the governance's side if there is more than 1 epoch for which rewards are already set. That is why I believe using the continue keyword is much better for ease of operation. Additionally, an event can be emitted for an epoch for which rewards are already set in order to make the governance aware of them. **(Note: This event emission is necessary as voters are expecting that the rewards will be set for the whole range - as mentioned by the sponsor above)**

Instead of this:

```
File: src/LendingLedger.sol
188:     function setRewards(
189:         uint256 _fromEpoch,
190:         uint256 _toEpoch,
191:         uint248 _amountPerEpoch
192:     ) external is_valid_epoch(_fromEpoch) is_valid_epoch(_toEpoch)
onlyGovernance {
193:         for (uint256 i = _fromEpoch; i <= _toEpoch; i += WEEK) {
194:             RewardInformation storage ri = rewardInformation[i];
195:             require(!ri.set, "Rewards already set");
196:             ri.set = true;
197:             ri.amount = _amountPerEpoch;
198:         }
199:     }
```

Use this (changes made on lines 195,196,197,198):

```
File: src/LendingLedger.sol
188:     function setRewards(
189:         uint256 _fromEpoch,
190:         uint256 _toEpoch,
191:         uint248 _amountPerEpoch
192:     ) external is_valid_epoch(_fromEpoch) is_valid_epoch(_toEpoch)
onlyGovernance {
193:         for (uint256 i = _fromEpoch; i <= _toEpoch; i += WEEK) {
194:             RewardInformation storage ri = rewardInformation[i];
195:             if(!ri.set) {
196:               emit RewardAlreadySetForEpoch(i); //@audit this can
make the sponsor aware of the epochs for which rewards are already set
before continuing
197:               continue;
198:             }
199:             ri.set = true;
200:             ri.amount = _amountPerEpoch;
201:         }
202:     }
```

[N-01] Public variables only used within contract can be changed to private visibility

Public variables that are only used within the contract can be marked private. There are 30 instances of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L26C1-L28C34

There are 3 instances here:

```
File: src/VotingEscrow.sol
26:     string public name;
27:     string public symbol;
28:     uint256 public decimals = 18;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L36C1-L41C53

There are 6 instances here:

```
File: src/VotingEscrow.sol
36:     uint256 public globalEpoch;
37:     Point[100000000000000000] public pointHistory; // 1e9 *
userPointHistory-length, so sufficient for 1e9 users
38:     mapping(address => Point[1000000000]) public userPointHistory;
39:     mapping(address => uint256) public userPointEpoch;
```

```
40:     mapping(uint256 => int128) public slopeChanges;
41:     mapping(address => LockedBalance) public locked;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L13C1-L27C1
https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L32

There are 9 instances here:

```
File: src/LendingLedger.sol
13:     address public governance;
14:     GaugeController public gaugeController;
15:     mapping(address => bool) public lendingMarketWhitelist;
16:     /// @dev Lending Market => Lender => Epoch => Balance
17:     mapping(address => mapping(address => mapping(uint256 =>
uint256))) public lendingMarketBalances; // cNote balances of users within
the lending markets, indexed by epoch
18:     /// @dev Lending Market => Lender => Epoch
19:     mapping(address => mapping(address => uint256)) public
lendingMarketBalancesEpoch; // Epoch when the last update happened
20:     /// @dev Lending Market => Epoch => Balance
21:     mapping(address => mapping(uint256 => uint256)) public
lendingMarketTotalBalance; // Total balance locked within the market, i.e.
sum of lendingMarketBalances for all
22:     /// @dev Lending Market => Epoch
23:     mapping(address => uint256) public lendingMarketTotalBalanceEpoch;
// Epoch when the last update happened
24:
25:     /// @dev Lending Market => Lender => Epoch
26:     mapping(address => mapping(address => uint256)) public
userClaimedEpoch; // Until which epoch a user has claimed for a particular
market (exclusive this value)
32:     mapping(uint256 => RewardInformation) public rewardInformation;
```

https://github.com/code-423n4/2023-08-verwa/blob/9a2e7be003bc1a77b3b87db31f3d5a1bcb48ed32/src/GaugeController.sol#L24C1-L38C1

There are 12 instances here:

```
File: src/GaugeController.sol
24:     VotingEscrow public votingEscrow;
25:     address public governance;
26:     mapping(address => bool) public isValidGauge;
27:     mapping(address => mapping(address => VotedSlope)) public
vote_user_slopes;
28:     mapping(address => uint256) public vote_user_power;
29:     mapping(address => mapping(address => uint256)) public
last_user_vote;
```

```
30:
31:     mapping(address => mapping(uint256 => Point)) public
points_weight;
32:     mapping(address => mapping(uint256 => uint256)) public
changes_weight;
33:     mapping(address => uint256) time_weight;
34:
35:     mapping(uint256 => Point) points_sum;
36:     mapping(uint256 => uint256) changes_sum;
37:     uint256 public time_sum;
```

[N-02] Variables that are unchanging should be marked constant or immutable if assigned in constructor

Variables than do not change can be marked constant and those that do not change after assignment during construction time can be marked immutable There are 7 instances of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L26C1-L28C34

There are 3 instances here: Vsriables name and symbol can be marked immutable and decimals can be marked constant.

```
File: src/VotingEscrow.sol
26:     string public name;
27:     string public symbol;
28:     uint256 public decimals = 18;
```

https://github.com/code-423n4/2023-08-verwa/blob/9a2e7be003bc1a77b3b87db31f3d5a1bcb48ed32/src/GaugeController.sol#L24C1-L25C31

There are 2 instances here: Variables can be made immutable since they do not change after construction time.

```
File: src/GaugeController.sol
24:     VotingEscrow public votingEscrow;
25:     address public governance;
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L13C1-L15C1

There are 2 instances here: Variables can be made immutable since they do not change after construction time.

```
File: src/LendingLedger.sol
13:     address public governance;
```

```
14:     GaugeController public gaugeController;
```

[N-03] Missing event emission for critical storage changes in functions

Functions that make storage updates or critical configuration updates are missing events and/or their emissions. There are 11 instances of this:

1. _checkPoint()
2. _checkpoint_lender()
3. _checkpoint_market()
4. checkpoint_market()
5. checkpoint_lender()
6. sync_ledger()
7. claim()
8. setRewards()
9. whiteListLendingMarket()
10. _change_gauge_weight()
11. vote_for_gauge_weights()

[N-04] Remove redundant code and comments to improve code readability and maintainability

Redundant code logic and comments serving no purpose in the codebase can be removed to improve code readability and maintainability.

There are 6 instances of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L212

We do not need to check if `_user_weight >= 0` in the require statement below since the input `_user_weight` is taken as an uint256 parameter, thus it will always be greater than or equal to 0. If the function is called with `_user_weight` less than zero, the function reverts automatically due to it not being an uint256. Thus, the first check can be removed.

```
File: src/GaugeController.sol
212: require(_user_weight >= 0 && _user_weight <= 10_000, "Invalid user
weight");
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L142C1-L144C14

The if block below can be removed since Line 143 is being overwritten on Line 149 here. The uEpoch is 0 only when the user is creating a lock for the first time. Thus, since it is being overwritten anyway, the if block can be removed.

```
File: src/VotingEscrow.sol
142:             if (uEpoch == 0) {
143:                 userPointHistory[_addr][uEpoch + 1] = userOldPoint;
144:             }
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L340

Line 340 below can be removed since it serves no purpose in the code. It is not related to the call to _checkPoint() on Line 344 as well.

```
File: src/VotingEscrow.sol
340:  newLocked.delegated = 0;
```

Note: Although `_checkpoint` theoretically reads it below, `_newLocked.end > block.timestamp` is never true in this case (as the end is 0 since it was updated in the withdraw() function), thus should be safe to remove

```
            if (_newLocked.end > block.timestamp && _newLocked.delegated >
0) {
                userNewPoint.slope = _newLocked.delegated /
int128(int256(LOCKTIME));
                userNewPoint.bias = userNewPoint.slope *
int128(int256(_newLocked.end - block.timestamp));
            }
```

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

TODO statement should be solved and removed.

```
File: src/LendingLedger.sol
48: governance = _governance; // TODO: Maybe change to Oracle
```

https://github.com/code-423n4/2023-08-verwa/blob/9a2e7be003bc1a77b3b87db31f3d5a1bcb48ed32/src/GaugeController.sol#L59

TODO statement should be solved and removed.

```
File: src/GaugeController.sol
59: governance = _governance; // TODO: Maybe change to Oracle
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L4

Import can be removed since it is not used anywhere in the LendingLedger.sol contract.

```
File: src/LendingLedger.sol
4: import {VotingEscrow} from "./VotingEscrow.sol";
```

[N-05] Consider using delete instead of assigning address(0) to clear values

(Note: This instance is missing in the [N-31] bot finding)

The delete keyword more closely matches the semantics of what is being done (deletion of lock), and draws more attention to the changing of state, which may lead to a more thorough audit of its associated logic.

There is 1 instance of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/VotingEscrow.sol#L338

```
File: src/VotingEscrow.sol
338: newLocked.delegatee = address(0);
```

[N-06] Require statements should provide an error message in case condition fails

An error message should be added to provide meaning on failure of condition in require statements.

There are 2 instances of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/LendingLedger.sol#L42

A short error message like `!auth` should be provided for clarity on failure.

```
File: src/LendingLedger.sol
42: require(msg.sender == governance);
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L51

A short error message like `!auth` should be provided for clarity on failure.

```
File: src/GaugeController.sol
51: require(msg.sender == governance);
```

[N-07] Modify code to improve code readability and maintainability

Code should be refactored to improve code readability and maintainability. This removes any additional redundant code as well.

There is 1 instance of this issue:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L60C1-L62C1

Instead of this:

```
File: src/GaugeController.sol
60:         uint256 last_epoch = (block.timestamp / WEEK) * WEEK;
61:         time_sum = last_epoch;
```

Use this:

```
File: src/GaugeController.sol
60:         time_sum = (block.timestamp / WEEK) * WEEK;
```

[N-08] Return values not handled in functions

There are 5 instances here:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L180C1-L181C20

The _get_weight() and _get_sum() functions return uint256 values. But they are not handled here. There are 2 instances here:

```
File: src/GaugeController.sol
180:         _get_weight(_gauge);
181:         _get_sum();
```

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

_get_sum() returns an uint256 value, which is not handled here. There is 1 instance here:

```
File: src/GaugeController.sol
136: _get_sum();
```

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L142C1-L143C20

The _get_weight() and _get_sum() functions return uint256 values. But they are not handled here. There are 2 instances here:

```
File: src/GaugeController.sol
142:         _get_weight(_gauge);
143:         _get_sum();
```

[N-09] Duplicated require/if statements should be refactored

(Note: The instance below is not included in the [N-10] bot finding)

There is 1 instance of this:

https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L128
https://github.com/code-423n4/2023-08-verwa/blob/a693b4db05b9e202816346a6f9cada94f28a2698/src/GaugeController.sol#L213C69-L213C69

```
File: src/GaugeController.sol
128: require(isValidGauge[_gauge], "Invalid gauge address"); //@audit
Duplicate on Line 213
213: require(isValidGauge[_gauge_addr], "Invalid gauge address");
```

Analysis Report

Comments for the judge to contextualize findings

My issues comprise 2 Highs, 1 Medium, 9 Non-Critical and 5 Low issues. Below listed are some important contextual points on both my high severity findings, 1 Medium and 1 low severity finding.

1. Root cause of both my highs are different - The impact of both the high risk findings is the same (i.e. locking of user's funds) but the root cause is different. The first issue arises due to the user's end time being shorter than the previous assigned delegatee's. The second issue arises due to the user's lock expiring under an existing delegation to another delegatee. Why both these issues look similar (though they're not) is due to the user not being able to transfer delegation back to themselves. Even if we solve the first issue, the second issue still persists. This I believe is proof enough for them to be considered as separate issues.

2. My Medium issue challenges the `LOCKTIME` of 5 years, which is native to the project idea. Due to this, an issue arises where the user does not receive any voting power on deposits below 1825 CANTO. I've added more context on this issue in the following sections below. Additionally, this issue should be specified in the documentation for security researchers in future audits and made aware to the user by mentioning that a minimum deposit of 1825 CANTO is required to receive voting power.

3. One of my low severity issues - `[L-05] Consider skipping the epoch for which rewards are already set` - includes my discussions with the sponsor to provide more context on why I think my solution would be more useful for the governance/voters.

Approach taken in evaluating the codebase

The scope included only 3 files, therefore it was easy to identify the base to child contract path. It as follows:

```
VotingEscrow.sol (parentA) <= GaugeController.sol (childA parentB) <=
LendingLedger.sol (childA childB)
```

Day 1 : Audited VotingEscrow.sol with 413 SLOC (heaviest contract)

- Targeted important functions like createLock(), increaseAmount(), delegate(), withdraw() and _checkPoint() to understand how user locks are created, amounts for existing locks can be increased, delegations are done, withdrawal process and history for each of these functionalities is recorded time-to-time.
- Ensured all storage updates are correct.
- Followed along the happy case of creating a lock, increasing amount, delegating power and withdrawing tokens to spot issues. This is where I found 2 High issues related to delegation and withdrawals.
- Created written and coded POCs for the 2 issues spotted.
- Added inline bookmarks to QA issues spotted while reading the code.

Day 2 : Audited GaugeController.sol and LendingLedger.sol with 336 SLOC combined (lighter contracts)

- Targeted important functions in the GaugeController.sol and LendingLedger.sol such as vote_for_gauge_weights() and claim(). Rest of the functions were mainly checkpoint and syncing ledger related where I ensured all storage updates are correct.
- There was a lot of going back and forth between tests and the code to further ensure that storage updates were correct and aligned with the function behaviour.
- Noted down some architecture recommendations for the setRewards() function.
- Added inline bookmarks to QA issues spotted while reading the code.

Day 3 : Creating reports and proving validity of findings through tests

- Researching the impact of certain low severity issues.
- Created QA report
- Created Analysis Report

During all 3 days, I had several discussions with the sponsor on the architecture of the codebase (which I have documented in one of my low severity findings), especially because it is an implementation of the FIATDAO VotingEscrow contract with certain functions removed and modifications made.

Architecture recommendations

The sponsor has kept the codebase one step ahead in terms of security by implementing an already audited FIATDAO codebase. But the modifications made introduce two high severity and 1 Medium severity security bugs. Here is how:

- First modification: The `increaseUnlockTime()` function has been removed in the VotingEscrow.sol contract while the FIATDAO VE contract includes it. After following the happy case and spotting the 2 high issues, I believe this increaseUnlockTime() function is important and should be added with **access control** to the codebase as it will allow the person with access to extend a user's unlock time to a certain extent, allowing them to withdraw their locked CANTO.

- Second modification: The `LOCKTIME` for a user's lock has been fixed to 5 years, which is native to the project's idea I believe. But this introduces a higher entry barrier to those who want to create locks. To prevent the rounding issue (as mentioned in my issue #299), the user needs to deposit atleast 1825 CANTO to have valid voting power greater than 0. **(Note: The barrier might look low right now since we're in a bear market and 1825 CANTO is around 200 USD at the time of reading. But as we enter the bull market, the barrier becomes higher).** We could say that the barrier is proportional to the price of 1825 CANTO. Even if this LOCKTIME is native to the project idea, I would recommend the sponsor to reconsider it.

Other than these modifications, there is another change I would recommend in the architecture of the setRewards() function:

1. `[L-05] Consider skipping the epoch for which rewards are already set` - This finding is included in my QA report but it is worth mentioning here as well. The governance sets the rewards for epochs in the range X to Y, but if there is an epoch [X+1] in this range for which rewards are already set, all the previous changes made revert. Although this reversion is the desired behaviour (i.e. what the sponsor expects - discussed with sponsor) and can be solved with two separate proposals to go around the epoch [X+1] (rewards for which are already set) in the manner [X,X] and [X+2,Y], I think it might be time consuming and a bit heavy work-wise on the governance's side **if there is more than 1 epoch for which rewards are already set**. That is why I believe using the continue keyword is much better for ease of operation. Additionally, an event can be emitted for an epoch for which rewards are already set in order to make the governance aware of them. (Note: This event emission is necessary as voters are expecting that the rewards will be set for the whole range - as mentioned by the sponsor)

Codebase quality analysis

The codebase quality is between above average. Why not Medium or High you may ask? I say this due to 2 reasons:

1. The test coverage is above 90%, which puts the codebase one step ahead in terms of security. Due to this, I would assign it the High rank.
2. If one follows the happy case for the process of delegation, the issue of not being able to transfer delegation back to the user has not been tested. Such issues arising in the happy case itself should be thoroughly tested first as it is fundamental to the functioning of the protocol. Due of this, I would assign it the Low rank as the happy case or user flow should be tested first.
3. The sponsor has done an amazing job with storage updates in the increaseAmount() function. This is because if you observe the previous findings in the FIATDAO C4 contest, one can observe how unlock times could be increased on expired locks. This is well mitigated by the sponsor by adding a

require statement to check for lock expiry and updating the `newLocked.amount` and `newLocked.delegated` storage variables correctly. Due to this, I would assign it the High rank.

Centralization risks

- There is some degree of centralisation introduced through the governance/voters.
- The implementation of the governance mechanism is out of scope but it is important to ensure it is implemented correctly since there are several configuration functions like add_gauge(), remove_gauge(), whitelisting lender markets and setting epoch rewards, which are critical to the functioning of the codebase.

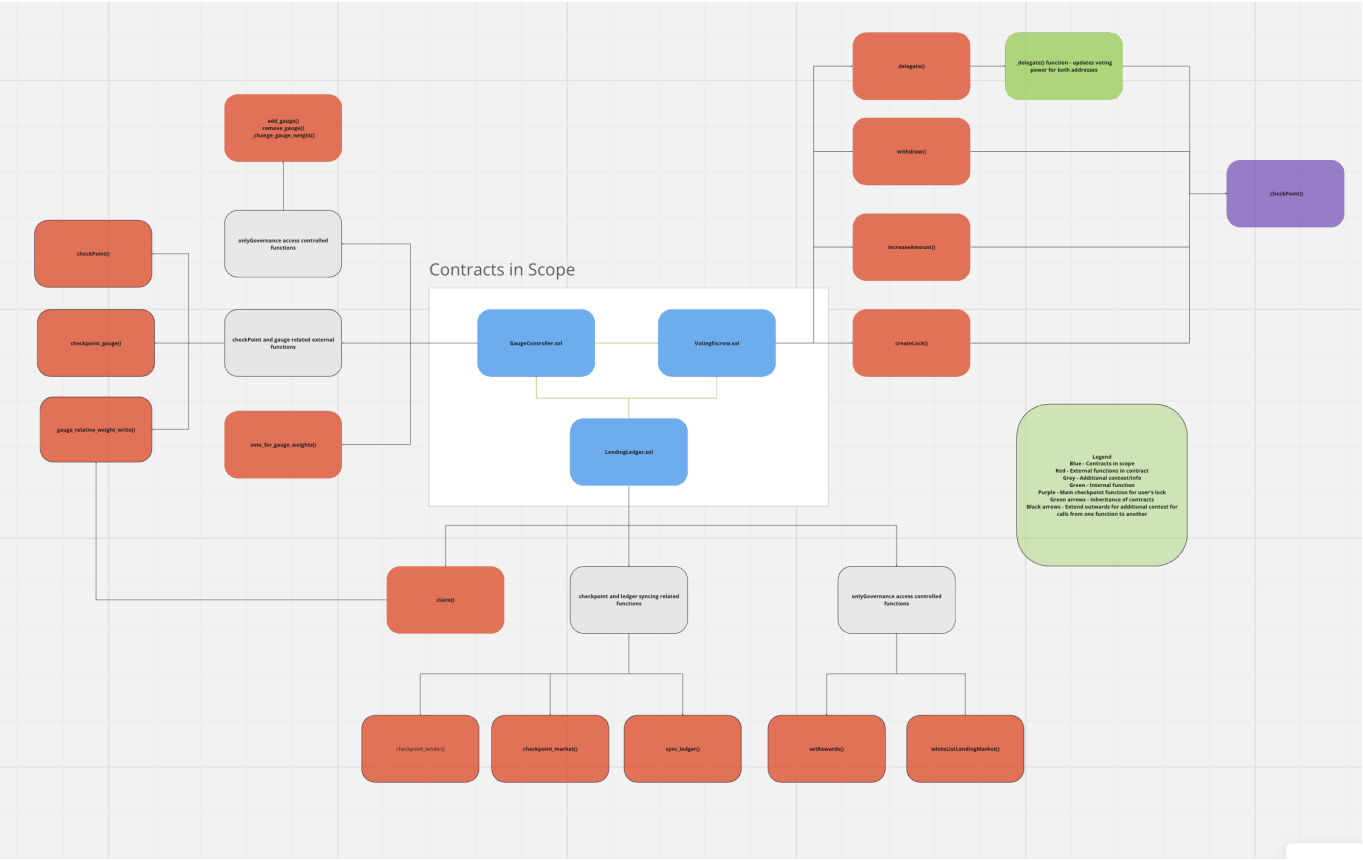
Mechanism review

Gauge model used: Linear decay

Function/Equation for this decaying linear model: W(t) = m*t + b , where b is the bias and m is the slope, and t is the time.

Brief explanation of the model: A user can lock up tokens and receive voting power. User's voting power decays linearly over time (in our case over 5 years, i.e. is 0 after 5 years). The VotingEscrow.sol stores the linear function (which is parameterized by a slope and a bias) of this user and then you can always linearly interpolate the time if amount for lock is increased by increaseAmount() function.

Here is a mindmap created to get an idea of the important contract and their function flow: https://drive.google.com/file/d/1RQ3WPYnjOEBlg1FDJfJUZDjwhN1Xl1Ua/view?usp=sharing



Time spent:

30 hours