// Smart Contract Security Assessment 09.16.2024 - 09.27.2024

VE Governance Hub

Aragon

HALBORN

VE Governance Hub - Aragon

Prepared by: HALBORN

Last Updated 10/01/2024

Date of Engagement by: September 16th, 2024 - September 27th, 2024

Summary

86% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
22	1	5	2	2	12

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
    - 7.1 Token id reuse leads to protocol deadlock
    - 7.2 Inaccurate voting power reporting after withdrawal initiation
    - 7.3 Unsafe minting in lock contract
    - 7.4 Critical withdrawal blockage due to escrow address whitelisting
    - 7.5 Potential token lock due to unrestricted escrow transfers
    - 7.6 Unrestricted nft contract change enables potential fund theft
    - 7.7 Checkpoint function allows non-chronological updates
    - 7.8 Incorrect balance tracking for non-standard erc20 tokens
    - 7.9 Potential bypass of minimum lock time at checkpoint boundaries
    - 7.10 Shared role for pause and unpause functions
    - 7.11 Inconsistent voting power reporting in reset event
    - 7.12 Critical roles not assigned in contract setup
    - 7.13 Lack of contract validation in initializer
    - 7.14 Unbounded warmup period can span multiple epochs

7.15 Misleading variable name for user interaction tracking
7.16 Unset dependencies may cause reverts and improper behavior
7.17 Redundant exit check and missing documentation in exit function
7.18 Fee precision too high
7.19 Unnecessary calculation in checkpoint function
7.20 Potentially unnecessary check in reset function
7.21 Inconsistent behavior in epoch start calculation
7.22 Misleading constant name for voting period
8. Automated Testing

1. Introduction

Aragon engaged our security analysis team to conduct a comprehensive security audit of their smart contract ecosystem. The primary aim was to meticulously assess the security architecture of the smart contracts to pinpoint vulnerabilities, evaluate existing security protocols, and offer actionable insights to bolster security and operational efficacy of their smart contract framework. Our assessment was strictly confined to the smart contracts provided, ensuring a focused and exhaustive analysis of their security features.

2. Assessment Summary

Our engagement with Aragon spanned a 2 week period, during which we dedicated one full-time security engineer equipped with extensive experience in blockchain security, advanced penetration testing capabilities, and profound knowledge of various blockchain protocols. The objectives of this assessment were to:

- Verify the correct functionality of smart contract operations.

- Identify potential security vulnerabilities within the smart contracts.

- Provide recommendations to enhance the security and efficiency of the smart contracts.

3. Test Approach And Methodology

Our testing strategy employed a blend of manual and automated techniques to ensure a thorough evaluation. While manual testing was pivotal for uncovering logical and implementation flaws, automated testing offered broad code coverage and rapid identification of common vulnerabilities. The testing process included:

- A detailed examination of the smart contracts' architecture and intended functionality.

- Comprehensive manual code reviews and walkthroughs.

- Functional and connectivity analysis utilizing tools like Solgraph.

- Customized script-based manual testing and testnet deployment using Foundry.

This executive summary encapsulates the pivotal findings and recommendations from our security assessment of Aragon smart contract ecosystem. By addressing the identified issues and implementing the recommended fixes, Aragon can significantly boost the security, reliability, and trustworthiness of its smart contract platform.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: ve-governance

(b) Assessed Commit ID: 98912bf

(c) Items in scope:

- src/voting/SimpleGaugeVoter.sol
- src/voting/SimpleGaugeVoterSetup.sol
- src/voting/ISimpleGaugeVoter.sol
- src/libs/SignedFixedPointMathLib.sol
- src/libs/CurveConstantLib.sol
- src/libs/ProxyLib.sol
- src/escrow/increasing/Lock.sol
- src/escrow/increasing/ExitQueue.sol
- src/escrow/increasing/VotingEscrowIncreasing.sol
- src/escrow/increasing/QuadraticIncreasingEscrow.sol
- src/escrow/increasing/interfaces/IEscrowCurveIncreasing.sol
- src/escrow/increasing/interfaces/IExitQueue.sol
- src/escrow/increasing/interfaces/IVotingEscrowIncreasing.sol
- src/escrow/increasing/interfaces/IERC721EMB.sol
- src/escrow/increasing/interfaces/IVotes.sol
- src/escrow/increasing/interfaces/ILock.sol
- src/clock/Clock.sol
- src/clock/IClock.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- https://github.com/aragon/ve-governance/pull/3
- https://github.com/aragon/ve-governance/pull/22
- https://github.com/aragon/ve-governance/pull/4
- https://github.com/aragon/ve-governance/pull/5
- https://github.com/aragon/ve-governance/pull/6
- https://github.com/aragon/ve-governance/pull/7
- https://github.com/aragon/ve-governance/pull/26
- https://github.com/aragon/ve-governance/pull/17
- https://github.com/aragon/ve-governance/pull/25
- https://github.com/aragon/ve-governance/pull/16
- https://github.com/aragon/ve-governance/pull/11

- https://github.com/aragon/ve-governance/pull/27
- https://github.com/aragon/ve-governance/pull/15
- https://github.com/aragon/ve-governance/pull/14
- https://github.com/aragon/ve-governance/pull/12
- https://github.com/aragon/ve-governance/pull/13
- https://github.com/aragon/ve-governance/pull/10

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	5	2	2	12

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TOKEN ID REUSE LEADS TO PROTOCOL DEADLOCK	CRITICAL	SOLVED - 09/27/2024
INACCURATE VOTING POWER REPORTING AFTER WITHDRAWAL INITIATION	HIGH	FUTURE RELEASE
UNSAFE MINTING IN LOCK CONTRACT	HIGH	SOLVED - 09/27/2024
CRITICAL WITHDRAWAL BLOCKAGE DUE TO ESCROW ADDRESS WHITELISTING	HIGH	SOLVED - 09/27/2024
POTENTIAL TOKEN LOCK DUE TO UNRESTRICTED ESCROW TRANSFERS	HIGH	SOLVED - 09/27/2024
UNRESTRICTED NFT CONTRACT CHANGE ENABLES POTENTIAL FUND THEFT	HIGH	SOLVED - 09/27/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CHECKPOINT FUNCTION ALLOWS NON-CHRONOLOGICAL UPDATES	MEDIUM	SOLVED - 09/27/2024
INCORRECT BALANCE TRACKING FOR NON-STANDARD ERC20 TOKENS	MEDIUM	SOLVED - 09/27/2024
POTENTIAL BYPASS OF MINIMUM LOCK TIME AT CHECKPOINT BOUNDARIES	LOW	SOLVED - 09/27/2024
SHARED ROLE FOR PAUSE AND UNPAUSE FUNCTIONS	LOW	FUTURE RELEASE
INCONSISTENT VOTING POWER REPORTING IN RESET EVENT	INFORMATIONAL	SOLVED - 09/27/2024
CRITICAL ROLES NOT ASSIGNED IN CONTRACT SETUP	INFORMATIONAL	SOLVED - 09/27/2024
LACK OF CONTRACT VALIDATION IN INITIALIZER	INFORMATIONAL	ACKNOWLEDGED
UNBOUNDED WARMUP PERIOD CAN SPAN MULTIPLE EPOCHS	INFORMATIONAL	PARTIALLY SOLVED - 09/27/2024
MISLEADING VARIABLE NAME FOR USER INTERACTION TRACKING	INFORMATIONAL	SOLVED - 09/27/2024
UNSET DEPENDENCIES MAY CAUSE REVERTS AND IMPROPER BEHAVIOR	INFORMATIONAL	FUTURE RELEASE

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
REDUNDANT EXIT CHECK AND MISSING DOCUMENTATION IN EXIT FUNCTION	INFORMATIONAL	ACKNOWLEDGED
FEE PRECISION TOO HIGH	INFORMATIONAL	SOLVED - 09/27/2024
UNNECESSARY CALCULATION IN CHECKPOINT FUNCTION	INFORMATIONAL	SOLVED - 09/27/2024
POTENTIALLY UNNECESSARY CHECK IN RESET FUNCTION	INFORMATIONAL	SOLVED - 09/27/2024
INCONSISTENT BEHAVIOR IN EPOCH START CALCULATION	INFORMATIONAL	SOLVED - 09/27/2024
MISLEADING CONSTANT NAME FOR VOTING PERIOD	INFORMATIONAL	SOLVED - 09/27/2024

7. FINDINGS & TECH DETAILS

7.1 TOKEN ID REUSE LEADS TO PROTOCOL DEADLOCK

// CRITICAL

Description

The `createLock` function in the `VotingEscrow` contract generates token IDs based on the `totalSupply()` of the `lockNFT`. When a user locks tokens, they receive an NFT with an ID equal to the current `totalSupply()`. However, after a withdrawal, the `burn` operation reduces the `totalSupply()`, potentially leading to the reuse of previously minted token IDs. This creates a critical issue where the protocol reuses an old token ID that retains past voting power, causing conflicts and reverts.
This leads to two major problems:
1. Token ID Duplication: After a withdrawal, a new lock creation can reuse a previously used token ID, potentially inheriting outdated voting power data and causing conflicts in the protocol. Scenario:

- USER 1 creates a lock, receiving token ID 1.
- USER 1 initiates a withdrawal by calling `beginWithdrawal` for token ID 1.
- USER 1 completes the withdrawal using `withdraw` for token ID 1.
- USER 2 creates a lock, receiving token ID 1.
- USER 2 will have the old voting power as the same token ID was returned.

2. Protocol Deadlock: If a token ID is reused while another token with a higher ID still exists, the `createLock` function will revert due to the "token already minted" error, effectively blocking new lock creations and rendering the protocol unusable. Scenario:

- USER 1 creates a lock, receiving token ID 1.
- USER 2 creates a lock, receiving token ID 2.
- USER 1 initiates a withdrawal by calling `beginWithdrawal` for token ID 1.
- USER 1 completes the withdrawal using `withdraw` for token ID 1.
- USER 3 attempts to call `createLock`, but the transaction reverts with `ERC721: token already minted`.

Proof of Concept

Token ID re-usage:

```
function test_withdraw_duplicate_id() external {
    // mint
    token.mint(USER1, 100);

    vm.prank(USER1);
    token.approve(address(ve), 100);

    vm.prank(USER1);
    uint256 tokenId = ve.createLockFor(
        100,
```

```
        USER1
    ); // Token ID 1

    // Duplicated
    console.log("Token ID", tokenId);

    IVotingEscrow.LockedBalance memory locked = ve.locked(tokenId);
    console.log("Locked balance", locked.amount, locked.start);

    // approve
    vm.prank(USER1);
    lock.approve(address(ve), tokenId);

    vm.warp(1 weeks + 1 days);

    vm.prank(USER1);
    ve.beginWithdrawal(tokenId);

    // total supply
    console.log("Total supply", lock.totalSupply());

    ITicket.Ticket memory ticket = exitQueue.queue(tokenId);

    console.log("Ticket holder", ticket.holder);
    console.log("Ticket exit date", ticket.exitDate);

    vm.warp(ticket.exitDate);

    vm.prank(USER1);
    ve.withdraw(tokenId);

    // total supply
    console.log("Total supply", lock.totalSupply());

    vm.prank(USER1);
    token.approve(address(ve), 100);

    vm.prank(USER1);
    tokenId = ve.createLockFor(
        100,
        USER1
    ); // Token ID 1

    // Duplicated
    console.log("Token ID", tokenId);
}
```

Protocol deadlock:

```
function test_withdraw_duplicate_id_lock() external {

    // mint
    token.mint(USER1, 100);
    token.mint(USER2, 100);

    vm.prank(USER1);
    token.approve(address(ve), 100);

    vm.prank(USER2);
    token.approve(address(ve), 100);

    vm.prank(USER1);
    uint256 tokenId = ve.createLockFor(
        100,
        USER1
    ); // Token ID 1

    vm.prank(USER2);
    uint256 tokenId2 = ve.createLockFor(
        100,
        USER2
    ); // Token ID 2

    // Duplicated
    console.log("Token ID", tokenId);
    console.log("Token ID 2", tokenId2);

    IVotingEscrow.LockedBalance memory locked = ve.locked(tokenId);
    console.log("Locked balance", locked.amount, locked.start);

    // approve
    vm.prank(USER1);
    lock.approve(address(ve), tokenId);

    vm.warp(1 weeks + 1 days);

    vm.prank(USER1);
    ve.beginWithdrawal(tokenId);

    // total supply
    console.log("Total supply", lock.totalSupply());

    ITicket.Ticket memory ticket = exitQueue.queue(tokenId);
```

```
    console.log("Ticket holder", ticket.holder);
    console.log("Ticket exit date", ticket.exitDate);

    vm.warp(ticket.exitDate);

    vm.prank(USER1);
    ve.withdraw(tokenId);

    // total supply
    console.log("Total supply", lock.totalSupply());

    vm.prank(USER1);
    token.approve(address(ve), 100);

    vm.prank(USER1);
    tokenId = ve.createLockFor(
        100,
        USER1
    ); // Token ID 2 - Duplicated - Revert

    // Duplicated
    console.log("Token ID", tokenId);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:C/A:C/D:C/Y:N/R:N/S:C (10.0)

Recommendation

1. **Unique token ID generation**: Modify the `createLock` function to generate token IDs independently of the `totalSupply()`. One option is to maintain a separate counter or use a mapping that tracks the last used token ID and increments it with each new lock creation, ensuring that token IDs are never reused, even after withdrawals.
2. **Testing and validation**: Ensure comprehensive testing of scenarios where users create locks, withdraw tokens, and create new locks to prevent reintroduction of the issue.

These changes ensure that:

- Token IDs are always unique and never reused.
- The protocol avoids deadlock situations caused by token ID conflicts.
- The system maintains a consistent state even after multiple deposit and withdrawal cycles.

Remediation

SOLVED: The contract is now using a unique ID named `lastLockId` which is disconnected from the token supply and always incremented. The first value will be `1` keeping the same logic as before.

Remediation Hash

https://github.com/aragon/ve-governance/pull/3

7.2 INACCURATE VOTING POWER REPORTING AFTER WITHDRAWAL INITIATION

// HIGH

Description

In the `VotingEscrow` contract, the `_checkpointClear` function uses `epochNextCheckpointTs` to set the checkpoint for when the voting power becomes zero. This approach shifts the checkpoint to the nearest future epoch, meaning the voting power will not correctly reflect a withdrawal event between the current time (`block.timestamp`) and the future checkpoint time. As a result, the `votingPowerAt` function will incorrectly report non-zero voting power for an NFT that has already been transferred to escrow for withdrawal. This approach leads to a critical oversight where a token continues to report voting power after the withdrawal process has been initiated. Specifically:

- The voting power remains non-zero between the actual withdrawal time (block.timestamp) and the next checkpoint timestamp.
- For total voting power calculations, it introduces inaccuracies, as the withdrawn token still contributes to the voting power during that period.
- The discrepancy could be exploited in systems that rely on real-time voting power, such as Front-end or system components, leading to unintended governance outcomes and display incorrect values

Proof of Concept

```
function test_checkpoint_withdrawal() external {
    uint256 TOKEN_ID = 1;
    uint256 startTime = 0;

    // timestamp of 3 days (in the middle of the first week)
    vm.warp(3 days);

    // Emulate a "CreateLockFor" without token complexity

    // This new "checkpoint" will be the first week (604800 seconds)
    startTime = clock.epochNextCheckpointTs();
    console.log("START TIME", startTime);
    _do_checkpoint(TOKEN_ID, 1e18, startTime);
    _print_user_point(TOKEN_ID, startTime);
    _print_user_epoch(TOKEN_ID);

    // We now advance 3 days into the "next week"
    vm.warp(1 weeks + 3 days);

    // We emulate a `_checkpointClear`, aka withdrawal
    // This new "checkpoint" will be the second week (1209600 seconds) as
    // the checkpointClear take the end of the "checkpoint" epoch always.
```

```
    startTime = clock.epochNextCheckpointTs();
    console.log("START TIME", startTime);
    _do_checkpoint(TOKEN_ID, 0, startTime);
    _print_user_point(TOKEN_ID, startTime);
    _print_user_epoch(TOKEN_ID);


    // If we now request the votingPowerAt for the end of the second week, it should be 0
    console.log("VOTING POWER", qes.votingPowerAt(TOKEN_ID, startTime));
    assertEq(qes.votingPowerAt(TOKEN_ID, startTime), 0);

    // We have requested a withdrawal at 1 week + 3 days.
    // This means that i will be expecting that at 1 week + 4 days, the voting power will

    vm.warp(1 weeks + 4 days);
    // If we now request the votingPowerAt at the current time, it should be 0
    console.log("VOTING POWER", qes.votingPowerAt(TOKEN_ID, block.timestamp));

    // This is NOT 0, it should be 0
    assertEq(qes.votingPowerAt(TOKEN_ID, block.timestamp), 0);

}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:M/R:N/S:U (8.8)

Recommendation

1. **Use** `block.timestamp` in `_checkpointClear`: Instead of aligning the checkpoint to the future `epochNextCheckpointTs`, use the current time (`block.timestamp`) to accurately reflect the withdrawal event at the time it occurs. This ensures that voting power immediately reflects the withdrawal without waiting for the next checkpoint.
2. **Optional burn queue mechanism**: As suggested, introduce a `burnQueue` that records tokens entering withdrawal. The `votingPowerAt` function can then check this queue and return zero voting power for tokens that are in the withdrawal process, providing an additional safeguard and ensuring the system reflects the correct state in real time.
3. **Testing and verification**: Rigorously test this solution with edge cases using off-cycle timestamps to ensure that the voting power is correctly reported during withdrawal scenarios. Ensure that the system continues to work efficiently without introducing gas issues related to handling multiple checkpoints.

These changes will ensure that:

- Voting power is immediately zeroed for practical purposes upon withdrawal initiation.
- The system maintains compatibility with future implementations of total voting power calculations based on fixed intervals.
- There's an additional layer of protection against potential rehypothecation of voting power.

Implement these changes carefully and thoroughly test to ensure all voting power calculations and related functionalities work as expected across various scenarios.

Remediation

PENDING: Replaces the warmup mapping with a checkpoint ts. This adds treaceability and saves gas by writing the timestamps to a single storage slot. To address this issue, we could also add a burndown period, this is not added yet. However, with this change, adding a burndown at a later stage when total voting power is added will be trivial.

Remediation Hash

https://github.com/aragon/ve-governance/pull/22

7.3 UNSAFE MINTING IN LOCK CONTRACT

// HIGH

Description

The `Lock` contract currently uses the `_mint` function from `ERC721`, which does not include safety checks to ensure that the recipient of the minted token can handle ERC721 tokens. If the recipient is a contract that does not implement the `onERC721Received` interface, the token could be permanently locked in the contract without any way to interact with it. This risk could result in lost or inaccessible NFTs for users, especially in cases where tokens are minted to contracts that are not designed to handle ERC721 tokens. Additionally, while the current minting mechanism by ERC721 ensures that token IDs are unique (as the minting process prevents duplicate token IDs), the absence of safety checks increases the risk of accidental or intentional token mismanagement when interacting with contract addresses.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:C (7.8)

Recommendation

1. **Use** `_safeMint` instead of `_mint`: Replace the `_mint` function with `_safeMint` in the `Lock` contract to ensure that if tokens are transferred to a contract, it correctly implements the `onERC721Received` interface. This prevents tokens from being locked in contracts that cannot handle them.
2. **Reentrancy protection**: Ensure that the entire Escrow contract is protected against reentrancy attacks by applying the `nonReentrant` modifier. This protects against any potential attack vectors opened by the `onERC721Received` hook during the safe mint process.
3. **Testing**: Perform tests to ensure that the new safe minting process does not introduce reentrancy vulnerabilities or disrupt expected behavior, especially when interacting with contracts that support the `onERC721Received` hook.

By implementing these changes, you can protect users from losing access to tokens due to non-compliant contract recipients and ensure that the minting process is secure.

Remediation

SOLVED: The contract is now using the `_safeMint` version and has some tests proving that a contract can receive the minted token.

Remediation Hash

https://github.com/aragon/ve-governance/pull/4

7.4 CRITICAL WITHDRAWAL BLOCKAGE DUE TO ESCROW ADDRESS WHITELISTING

// HIGH

Description

In the `Lock` contract, the `setWhitelisted` function allows changing the whitelist status of any address, including the escrow address. This presents a critical vulnerability where the escrow address could be removed from the whitelist, effectively blocking all withdrawals in the system. The `_transfer` function checks the whitelist status of the recipient, and if the escrow is not whitelisted, all transfers to it (which are necessary for withdrawals) will fail.
This vulnerability could lead to a complete lockdown of user funds, as users would be unable to initiate withdrawals by transferring their NFTs back to the escrow contract.

Proof of Concept

```
function test_lock_set_whitelisted() external {
    address ESCROW = address(0x2000);

    DAOMockTrue dao = new DAOMockTrue();

    lock.initialize(
        ESCROW, // Simulate the escrow contract
        "NAME",
        "SYMBOL",
        address(dao)
    );

    // Only escrow can mint
    vm.prank(ESCROW);
    lock.mint(USER1, 1);

    // Disable the whitelist for escrow address
    vm.prank(ADMIN);
    lock.setWhitelisted(ESCROW, false);

    // Simulate a "withdrawal"
    // Approve the escrow to spend the token ID 1
    vm.prank(USER1);
    lock.approve(ESCROW, 1);

    // Simulate transferring the token as if it was called by the escrow contract
    vm.prank(ESCROW);
    vm.expectRevert(IWhitelistErrors.NotWhitelisted.selector);
```

```
    lock.transferFrom(USER1, ESCROW, 1);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:C (7.8)

Recommendation

1. **Prevent** `escrow` address modification: Modify the `setWhitelisted` function to prevent changes to the whitelist status of the `escrow` address. This ensures that the `escrow` address is always able to receive tokens during withdrawals, protecting the integrity of the withdrawal process.
2. **Alternative approach – bypass whitelist check for** `escrow`: Instead of relying on the whitelist mapping for the `escrow` address, add a condition in the `_transfer` function to allow transfers to the `escrow` address regardless of its whitelist status. This can be implemented as:

```
if (_to == escrow || whitelisted[WHITELIST_ANY_ADDRESS] || whitelisted[_to]) {
    // proceed with the transfer
}
```

1. **Testing**: Ensure tests are in place to verify that the `escrow` address cannot be removed from the whitelist and that all transfer scenarios involving the `escrow` address (such as withdrawals) function as expected.

By implementing this safeguard, you protect the system from the risk of disabling the `escrow` address, which could otherwise lead to blocked withdrawals and locked tokens.

Remediation

SOLVED: The code prevents the `setWhitelisted` call when the address is the escrow.

Remediation Hash

https://github.com/aragon/ve-governance/pull/5

7.5 POTENTIAL TOKEN LOCK DUE TO UNRESTRICTED ESCROW TRANSFERS

// HIGH

Description

In the `Lock` contract, allowing anyone to transfer their NFT directly to the `escrow` address, without an associated triggering function (such as `beginWithdrawal`), presents a serious issue. If a user mistakenly transfers their token directly to the `escrow`, there is no recovery mechanism to initiate the withdrawal process or move the token elsewhere. As a result, the NFT can become permanently stuck in the `escrow` contract, creating a situation where neither the owner nor the system can recover or withdraw the token.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:C (7.8)

Recommendation

1. **Restrict direct transfers to `escrow`**: Ensure that tokens can only be transferred to the `escrow` address through authorized contract calls that are part of the withdrawal process (e.g., `beginWithdrawal`). This can be enforced by modifying the `_transfer` function to check whether the transfer is part of an authorized action and not a direct user-initiated transfer.
2. **Implement a recovery mechanism**: Introduce a recovery mechanism within the `escrow` contract that allows the DAO or another trusted entity to recover tokens that are directly transferred to the `escrow` without going through the expected withdrawal flow. This could be a function like `recoverMistakenTransfer(uint256 tokenId)` that would allow a transfer of the token back to its rightful owner or another designated address.
3. **Testing**: Ensure testing covers edge cases where tokens are accidentally or maliciously transferred to the `escrow` address and validate that recovery mechanisms or restrictions function correctly.

By adding these safeguards, you prevent tokens from being trapped in the `escrow` and ensure that all token movements are properly accounted for and recoverable.

Remediation

SOLVED: The contract now includes a `sweepNFT` function, which enables tokens sent to the escrow contract but not involved in a withdrawal process to be transferred back to any specified address. It is the responsibility of the `SWEEPER_ROLE` to ensure the tokens are sent to the appropriate address.

Remediation Hash

https://github.com/aragon/ve-governance/pull/6

7.6 UNRESTRICTED NFT CONTRACT CHANGE ENABLES POTENTIAL FUND THEFT

// HIGH

Description

In the `VotingEscrow` contract, the `setLockNFT` function allows the DAO to change the NFT contract address without any restrictions. This functionality, while potentially useful for upgrades, introduces a critical vulnerability. An attacker with DAO privileges could exploit this by:
1. Creating a new malicious NFT contract with identical token IDs.
2. Changing the `lockNFT` address to point to this new contract.
3. Performing withdrawals for all tokens, effectively draining the entire contract balance.
This vulnerability exists because the contract assumes the integrity of the NFT contract and doesn't validate ownership transfers when changing the NFT contract address.

BVSS

AO:A/AC:L/AX:H/C:N/I:C/A:C/D:H/Y:C/R:N/S:C (7.0)

Recommendation

To mitigate this high-risk vulnerability, implement the following changes:
1. Remove the `setLockNFT` function entirely, making the NFT contract address immutable after initialization.
2. Set the `lockNFT` address during contract initialization.
3. Remove any functions or logic that allow changing the `lockNFT` address after initialization.
4. If future upgrades to the NFT contract are absolutely necessary, they should be handled through a carefully planned upgrade process using the contract's UUPS upgradeability feature, rather than a simple address change.
By making these changes, you eliminate the risk of unauthorized changes to the NFT contract address, significantly enhancing the security of the locked funds. This approach aligns with the principle of immutability for critical contract parameters and reduces the attack surface of the contract.

Remediation

SOLVED: The `setLockNFT` is only allowed to be called once.

Remediation Hash

https://github.com/aragon/ve-governance/pull/7

7.7 CHECKPOINT FUNCTION ALLOWS NON-CHRONOLOGICAL UPDATES

// MEDIUM

Description

In the `QuadraticIncreasingEscrow` contract, the `_checkpoint` function does not enforce chronological ordering of checkpoints. It allows the creation of a new checkpoint with a `_newLocked.start` value that could be less than the timestamp of the last recorded checkpoint. This can lead to the creation of a new epoch with a lower timestamp than the previous one, potentially causing issues with the binary search algorithm used in `_getPastUserPointIndex` and affecting the accuracy of historical voting power queries.

The `_getPastUserPointIndex` function, which relies on the chronological order of checkpoints, may return an invalid epoch due to this non-chronological update. This is because the binary search assumes a strictly increasing timestamp order, and the `_userPointHistory[_tokenId][_userEpoch].ts <= _timestamp` check may not behave as expected with out-of-order timestamps.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (6.3)

Recommendation

Implement a chronological check in the `_checkpoint` function to ensure that new checkpoints are always created with timestamps greater than or equal to the last recorded checkpoint.

These changes will help maintain the integrity of the checkpoint history and ensure more accurate historical voting power calculations.

Remediation

SOLVED: Code is correctly checking for timestamp to be equal or higher than any previous one.

Remediation Hash

https://github.com/aragon/ve-governance/pull/26

7.8 INCORRECT BALANCE TRACKING FOR NON-STANDARD ERC20 TOKENS

// MEDIUM

Description

The `VotingEscrow` contract does not account for tokens that have transfer fees, or tokens that are inflationary or deflationary. In the `createLockFor` function, the `_value` added to `totalLocked` and `LockedBalance` is based on the amount sent in the `safeTransferFrom`, without verifying the actual amount received by the contract. For tokens with fees, the deposited amount may be less than the transferred amount, causing discrepancies between what is stored in `LockedBalance` and the actual balance in the contract.

This could lead to incorrect calculations in the following scenarios: - Lock creation will overestimate the balance in `totalLocked` and `LockedBalance`. - Withdrawals may attempt to release more tokens than actually exist in the contract, leading to failures or potential underfunding.

Moreover, because the `safeTransferFrom` call occurs before the state changes, this also violates the check-effect-interaction pattern, exposing the contract to potential reentrancy vulnerabilities. Without proper reentrancy protection, a malicious token contract could exploit this by calling back into the contract during the token transfer.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:L/R:N/S:U (5.6)

Recommendation

1. **Transfer validation**: After calling `safeTransferFrom`, verify that the actual token balance change in the contract matches the expected `_value`. If there is a discrepancy (due to fees, inflation, or deflation), adjust the `totalLocked` and `LockedBalance` to reflect the actual received amount.
2. **Reentrancy protection**: Implement the `nonReentrant` modifier in all functions that involve token transfers or external interactions to mitigate reentrancy risks.
3. **Token compatibility checks**: Optionally, restrict the contract to only work with non-inflationary/deflationary tokens by adding a check that ensures the balance change after a transfer matches the expected amount. This can prevent the use of problematic tokens in the first place.

Remediation

SOLVED: Only tokens with 18 decimals and no transfer fees are supported. If the balance differs during the transfer the contract will revert.

Remediation Hash

https://github.com/aragon/ve-governance/pull/17

7.9 POTENTIAL BYPASS OF MINIMUM LOCK TIME AT CHECKPOINT BOUNDARIES

// LOW

Description

In the `ExitQueue` contract, the `queueExit` function prevents users from performing a deposit and withdrawal within the same checkpoint period. However, if a user deposits and queues a withdrawal at exactly the `epochNextCheckpointTs` (the start of the next epoch), they could bypass the intended locking period. This issue is exacerbated if the `minLock` value is set to 0, which would allow immediate withdrawal right after the deposit, breaking the locking mechanism's purpose.
This opens the protocol to potential abuse, where users can deposit and withdraw tokens without adhering to the expected lock periods, undermining the protocol's time-based locking guarantees.

BVSS

AO:A/AC:M/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (4.2)

Recommendation

1. **Enforce non-zero** `minLock`: Implement a check in the `setMinLock` function to ensure that `minLock` cannot be set to 0. This will enforce that there is always a minimum lock duration, preventing users from withdrawing tokens immediately after deposit.
2. **Adjust** `queueExit` logic: Consider adding additional checks in the `queueExit` function to ensure that even if a user deposits and attempts to withdraw at the exact start of the next epoch, they still adhere to the locking requirements.
3. **Testing and validation**: Rigorously test edge cases where users attempt to deposit and withdraw during the same checkpoint or epoch to ensure that no bypass of the locking mechanism is possible.
By enforcing a non-zero `minLock` and adjusting the `queueExit` logic, you can prevent users from exploiting the protocol to withdraw tokens immediately after depositing them. This strengthens the protocol's time-based locking mechanism and ensures fair use.

Remediation

SOLVED: `_setMinLock` cannot be called with a value of 0.

Remediation Hash

https://github.com/aragon/ve-governance/pull/25

7.10 SHARED ROLE FOR PAUSE AND UNPAUSE FUNCTIONS

// LOW

Description

In both the `VotingEscrow` and `SimpleGaugeVoter` contracts, the same role is used to both pause and unpause the contracts. This setup presents a risk: if an account with the pause privilege is compromised or misused, the same account could unpause the contract, potentially exacerbating malicious activity. Separating the roles for pausing and unpausing would provide a security layer that limits the impact of such events.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:H/D:N/Y:N/R:N/S:C (2.0)

Recommendation

Implement separate roles for `pause` and `unpause` functionalities. The `pause` function can remain with a specific role (such as an emergency role), but `unpause` should be restricted to the contract owner or a higher privilege role, such as `admin`. This separation reduces the risk of unintentional or malicious misuse of both functions by the same entity.

Remediation

PENDING: Noted for future releases

7.11 INCONSISTENT VOTING POWER REPORTING IN RESET EVENT

// INFORMATIONAL

Description

In the `SimpleGaugeVoter` contract, the `Reset` event currently emits `totalVotingPowerCast - _votes` as the total voting power for the gauge being reset. However, this differs from the approach used in the `Voted` event, which emits `gaugeVotes[gauge]` for the total voting power. While this discrepancy does not introduce a security vulnerability, it creates inconsistency in how the total voting power is reported between the `Voted` and `Reset` events.

This inconsistency may cause issues when aggregating or interpreting data off-chain, as the two events report voting power differently. Consistency between the `Voted` and `Reset` events can prevent potential confusion or bugs in off-chain systems.

BVSS

AO:S/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:C (1.9)

Recommendation

To ensure consistency with the `Voted` event, update the `Reset` event to use `gaugeVotes[gauge]` for the total voting power, as this reflects the same approach used when emitting the `Voted` event. This change will make it easier for off-chain systems to correctly interpret and aggregate voting power changes.

Before:

```
emit Reset({
    voter: _msgSender(),
    gauge: gauge,
    epoch: epochId(),
    tokenId: _tokenId,
    votingPower: _votes,
    totalVotingPower: totalVotingPowerCast - _votes,  // inconsistent with Voted event
    timestamp: block.timestamp
});
```

After:

```
emit Reset({
    voter: _msgSender(),
    gauge: gauge,
    epoch: epochId(),
    tokenId: _tokenId,
    votingPower: _votes,
    totalVotingPower: gaugeVotes[gauge],  // consistent with Voted event
```

```
    timestamp: block.timestamp
});
```

This update ensures consistency across events and helps prevent potential issues with off-chain data analysis or interpretation.

Remediation

SOLVED: The events have been cleaned and they are now consistent across all functions.

Remediation Hash

https://github.com/aragon/ve-governance/pull/16

7.12 CRITICAL ROLES NOT ASSIGNED IN CONTRACT SETUP

// INFORMATIONAL

Description

In the `SimpleGaugeVoterSetup` contract, several critical roles are not assigned during the setup process. Specifically:
1. The `WITHDRAW_ROLE` for the ExitQueue contract, which is necessary for withdrawing accumulated fees.
2. The `PAUSER_ROLE` for the VotingEscrow contract, which is crucial for emergency stops.
3. The `SWEEPER_ROLE` for the VotingEscrow contract, which is needed for recovering excess tokens.
The absence of these role assignments could lead to: - Inability to withdraw accumulated fees from the ExitQueue. - Lack of emergency pause functionality in critical situations. - Inability to recover excess tokens sent to the VotingEscrow contract.
These oversights could significantly impact the operational capabilities and security of the system.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:C (1.6)

Recommendation

1. **Assign** `WITHDRAW_ROLE` in setup: Ensure that the `WITHDRAW_ROLE` in the `ExitQueue` contract is assigned to an appropriate entity, such as the DAO or another trusted role, during the setup. This ensures that the system can withdraw fees when necessary.
2. **Assign** `PAUSER_ROLE` and `SWEEPER_ROLE` for `VotingEscrow`: Similarly, assign the `PAUSER_ROLE` and `SWEEPER_ROLE` to trusted entities during the setup process to ensure that pausing the contract (in case of emergencies) and sweeping excess funds is possible.
3. **Update** `SimpleGaugeVoterSetup` permissions: Modify the permissions setup process to include these role assignments.
4. **Testing**: Ensure that these roles are correctly assigned during the setup process and that they can be used as expected (e.g., withdrawing fees, pausing the contract, sweeping excess tokens).
By assigning these roles during setup, you ensure that the system retains its full administrative capabilities, avoiding potential issues like locked funds or the inability to manage the contract in emergency situations.

Remediation

SOLVED: Those roles have been assigned to the dao by default during setup.

Remediation Hash

https://github.com/aragon/ve-governance/pull/11

7.13 LACK OF CONTRACT VALIDATION IN INITIALIZER

// INFORMATIONAL

Description

In the `QuadraticIncreasingEscrow` and `VotingEscrow` contracts, there are no checks to ensure that the addresses provided during initialization conform to the expected interfaces, such as through `ERC165` or similar contract validation mechanisms. This omission assumes that the deployment contract will provide valid addresses. If an invalid address is passed, it could result in misbehavior or require redeployment, potentially incurring significant costs and operational delays.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (1.3)

Recommendation

Add `ERC165` interface checks during initialization to verify that the provided addresses conform to the expected contracts (such as `IVotingEscrow`, `IEscrowCurve`, etc.). This will ensure that only compatible contracts are used, reducing the risk of errors and redeployments.

Remediation

ACKNOWLEDGED: The DAO and deployers are responsible for ensuring they validate the contracts and can update if needed

7.14 UNBOUNDED WARMUP PERIOD CAN SPAN MULTIPLE EPOCHS

// INFORMATIONAL

Description

In the `QuadraticIncreasingEscrow` contract, the `warmupPeriod` is set without an upper bound. This lack of a maximum cap allows the warm-up period to potentially span across multiple epochs. Such a scenario could lead to unexpected behavior in the voting power calculation and distribution, as the warm-up might overlap with subsequent voting periods or epochs. This could result in inconsistencies in the voting power allocation and potentially impact the overall fairness of the voting system.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (1.3)

Recommendation

Implement a maximum cap for the `warmupPeriod` to ensure it does not exceed the duration of a single epoch. Additionally, add a similar check in the initialize function to ensure the initial warm-up period is also within bounds. This change will prevent potential issues arising from excessively long warm-up periods and maintain the integrity of the epoch-based voting system.

Remediation

PARTIALLY SOLVED: The `beginWithdrawal` function now checks if the token has any voting power. This means that if the warmup period spans across multiple epochs, it could lead to `votingPower` being reported as 0 until the warmup period expires. It will not be possible to withdraw until the voting power is accounted for. However, the suggestion of capping the `warmupPeriod` to a max value was not considered.

Remediation Hash

https://github.com/aragon/ve-governance/pull/27

7.15 MISLEADING VARIABLE NAME FOR USER INTERACTION TRACKING

// INFORMATIONAL

Description

In the `QuadraticIncreasingEscrow` contract, the variable `userEpoch` is used to track the number of interactions that change locked amounts for a user. However, the name "epoch" is misleading as it suggests a relation to the global epoch system, while in reality, it represents checkpoints of user interactions. This naming convention can lead to confusion for developers maintaining or interacting with the contract, potentially causing misunderstandings about the variable's purpose and usage.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (1.3)

Recommendation

Rename the `userEpoch` variable to `userCheckpoint` throughout the contract. This change should be applied consistently to all related functions and variables.
Additionally, update all references to this variable in comments, function names, and documentation to reflect the new naming convention. This change will make the code more self-explanatory and reduce the risk of misinterpretation, improving the overall maintainability and readability of the contract.

Remediation

SOLVED: Renamed `UserPoint` to `TokenPoint` and `userPointEpoch` to `tokenPointInterval` , updated natspec along the process.

Remediation Hash

https://github.com/aragon/ve-governance/pull/15

References

7.16 UNSET DEPENDENCIES MAY CAUSE REVERTS AND IMPROPER BEHAVIOR

// INFORMATIONAL

Description

In the `VotingEscrow` contract, several functions rely on external addresses (`lockNFT`, `curve`, `voter`) that may not be set during initialization, leading to reverts when these addresses are accessed as `0x00..00`. Specifically: - Functions like `isApprovedOrOwner`, `ownedTokens`, `createLockFor`, `beginWithdrawal`, and `withdraw` interact with `lockNFT`, which could cause the contract to revert if `lockNFT` is unset. - Functions like `votingPowerAt`, `totalVotingPowerAt`, and `createLockFor` require the `setCurve` function to be called to initialize the `curve` contract. - `resetVotesAndBeginWithdrawal` requires the `voter` address to be set via `setVoter`.
Without ensuring these dependencies are set, users or integrators could encounter unexpected reverts and failures during interactions with the contract.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (1.3)

Recommendation

1. **Initialization during deployment**: Ensure that `lockNFT`, `curve`, and `voter` are set during the contract's initialization process to prevent potential reverts.
2. **Access control checks**: Implement checks in the relevant functions (e.g., `require` statements) to ensure that these dependencies are set before allowing interactions with the contract.
3. **Consider setting all critical dependencies as part of the contract initialization** to reduce the risk of them being missed during setup, preventing runtime failures.

Remediation

PENDING: Noted for future: implement a graceful and idiomatic revert strategy.

7.17 REDUNDANT EXIT CHECK AND MISSING DOCUMENTATION IN EXIT FUNCTION

// INFORMATIONAL

Description

In the `ExitQueue` contract, the `exit` function is called during the `withdraw` phase after an explicit call to `canExit` has already been made. However, the `exit` function also contains an internal `canExit` check, resulting in a duplicated validation. While this does not directly introduce security vulnerabilities, it adds unnecessary overhead and redundancy, which can be avoided.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (1.3)

Recommendation

1. **Remove redundant** `canExit` check: Since the caller (e.g., `withdraw`) already verifies that the ticket can be exited using the `canExit` function, the internal `canExit` check in `exit` is redundant. Remove this check from the `exit` function to streamline the code and reduce unnecessary validation.
2. **Add a note in the** `natspec`: Update the `exit` function's `natspec` documentation to specify that the caller must verify the ticket can be exited before calling `exit`. This ensures that developers are aware of the expected flow and avoid misuse.
3. **Testing**: Ensure the function works correctly without the internal `canExit` check by verifying that only valid exits are allowed through the expected flow.

By removing the redundant check and clearly documenting the expected flow, you reduce unnecessary gas consumption and simplify the exit logic, ensuring cleaner and more efficient code.

Remediation

ACKNOWLEDGED: Accepted the redundancy - they prefer the check in place given the critical nature of the exit queue.

7.18 FEE PRECISION TOO HIGH

// INFORMATIONAL

Description

In the `ExitQueue` contract, `feePercent` is used to calculate exit fees and is set with a precision of `1e18`, which is significantly higher than typically needed for percentage calculations. Generally, a precision of `10000` is sufficient to handle percentages with two decimal points (e.g., 1% represented as `100`, 0.01% as `1`). The use of `1e18` for this purpose introduces unnecessary complexity and gas usage, as it adds an extra overhead for calculations without any practical benefit.

BVSS

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (1.3)

Recommendation

1. **Adjust** `feePercent` precision: Reduce the precision of `feePercent` from `1e18` to `10000`, which is standard for percentage calculations with two decimal precision. This will simplify the calculations and improve the contract's gas efficiency without losing any meaningful precision.
2. **Update fee calculation logic**: Ensure that any calculations using `feePercent` are updated accordingly to reflect the new precision.
3. **Testing**: Ensure that all fee-related calculations work correctly with the new precision, and test various fee scenarios to ensure accuracy.

By using a precision of `10000` for `feePercent`, you maintain sufficient accuracy while optimizing gas usage and simplifying the code.

Remediation

SOLVED: Max fee value was reduced to `10000` and a constant variable named `MAX_FEE_PERCENT` created to make the code consistent.

Remediation Hash

https://github.com/aragon/ve-governance/pull/14

7.19 UNNECESSARY CALCULATION IN CHECKPOINT FUNCTION

// INFORMATIONAL

Description

In the `QuadraticIncreasingEscrow` contract, the `_checkpoint` function calculates the initial bias for a new lock using `getBias(0, amount)` when `!isExiting`. This calculation is unnecessary because when the elapsed time is 0, the function will always return a value equal to the amount. This redundant calculation not only adds complexity to the code but also consumes more gas than necessary.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (0.6)

Recommendation

Simplify the bias calculation in the `_checkpoint` function by directly setting the bias equal to the amount when initializing a new lock. Replace the following lines:

```
if (!isExiting) {
    uNew.coefficients = getCoefficients(amount);
    // for a new lock, write the base bias (elapsed == 0)
    uNew.bias = getBias(0, amount);
}
```

with:

```
if (!isExiting) {
    uNew.coefficients = getCoefficients(amount);
    uNew.bias = amount;
}
```

This change will make the code more straightforward and efficient, reducing gas costs for users creating new locks. It also maintains the same functionality as the original implementation while improving code readability.

Remediation

SOLVED: Instead of setting it to the amount, the internal `_getBias` is used instead to be consistent.

Remediation Hash

https://github.com/aragon/ve-governance/pull/12

7.20 POTENTIALLY UNNECESSARY CHECK IN RESET FUNCTION

// INFORMATIONAL

Description

In the `_reset` function of the `SimpleGaugeVoter` contract, there is an `if (_votes != 0)` check before the code that removes votes for a given gauge. However, this check seems redundant because the `_vote` function already ensures that `votesForGauge != 0` to prevent double voting. While this additional check does not cause harm or introduce security risks, it adds unnecessary complexity and slightly increases gas costs.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (0.6)

Recommendation

Consider removing the redundant `if (_votes != 0)` check in the `_reset` function, as the logic in the `_vote` function already ensures that votes will not be zero. This simplifies the code and reduces gas usage, even though the impact is minimal.

Remediation

SOLVED: The conditional check was removed.

Remediation Hash

https://github.com/aragon/ve-governance/pull/13

7.21 INCONSISTENT BEHAVIOR IN EPOCH START CALCULATION

// INFORMATIONAL

Description

In the `Clock` contract, the `resolveEpochStartsIn` and `resolveEpochNextCheckpointIn` functions have a discrepancy between its behavior and its NatSpec description. The function is described as returning the "Number of seconds until the start of the next epoch". However, when the input timestamp is exactly at the start of an epoch, it returns 0 instead of the full duration until the next epoch. This inconsistency could lead to misinterpretation of the function's output, potentially causing issues in dependent code that assumes the function always returns a non-zero value for future epoch starts.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation

Consider updating the NatSpec comment to accurately reflect the function's behavior, clarifying that it returns 0 when the timestamp is exactly at an epoch start. Alternatively, if maintaining consistency with the current description is crucial, modify the function to return the full `EPOCH_DURATION` when the timestamp is at an epoch boundary. However, be cautious of potential impacts on other parts of the codebase that may rely on the current behavior. A thorough review of all dependencies should be conducted before making any changes to this core timing function.

Remediation

SOLVED: Natspec is stating that: "If exactly at the start of the epoch, returns 0".

Remediation Hash

https://github.com/aragon/ve-governance/pull/10

7.22 MISLEADING CONSTANT NAME FOR VOTING PERIOD

// INFORMATIONAL

Description

In the `Clock` contract, the constant `VOTE_WINDOW_OFFSET` is named in a way that does not accurately reflect its purpose. This constant represents a period within the voting window, specifically a "warm-up" period at the beginning of the voting phase, rather than an offset to the start of the voting period as the name suggests. The current naming convention could lead to misunderstandings about the constant's role in the voting process, potentially causing errors in implementation or future modifications to the contract.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation

Rename the `VOTE_WINDOW_OFFSET` constant to better reflect its actual purpose. A more appropriate name could be `VOTE_WARMUP_PERIOD` or `VOTE_INITIAL_BUFFER`. This change would make the code more self-explanatory and reduce the risk of misinterpretation. After renaming, ensure to update all references to this constant throughout the codebase and in any associated documentation to maintain consistency and clarity.

Remediation

SOLVED: The variable was renamed to a more meaningful name (`VOTE_WINDOW_BUFFER`).

Remediation Hash

https://github.com/aragon/ve-governance/pull/10

8. AUTOMATED TESTING

```
INFO:Detectors:
Clock.resolveElapsedInEpoch(uint256) (src/clock/Clock.sol#79-83) uses a weak PRNG: "timestamp % EPOCH_DURATION (src/clock/Clock.sol#81)"
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#weak-PRNG
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#116)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
ExitQueue.withdraw(uint256) (src/escrow/increasing/ExitQueue.sol#125-128) ignores return value by underlying.transfer(msg.sender,_amount) (src/escrow/increasing/ExitQueue.sol#127)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#101)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#116)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#101)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#120)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#101)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#121)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#101)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#122)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#101)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#123)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#101)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#124)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#101)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#125)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-134) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#104)
        - result = prod0 * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#131)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Clock.resolveEpochNextCheckpointIn(uint256) (src/clock/Clock.sol#193-201) uses a dangerous strict equality:
        - elapsed == 0 (src/clock/Clock.sol#198)
Clock.resolveEpochStartsIn(uint256) (src/clock/Clock.sol#90-95) uses a dangerous strict equality:
        - (elapsed == 0) (src/clock/Clock.sol#93)
QuadraticIncreasingEscrow._getPastUserPointIndex(uint256,uint256) (src/escrow/increasing/QuadraticIncreasingEscrow.sol#221-249) uses a dangerous strict equality:
        - userPoint.ts == _timestamp (src/escrow/increasing/QuadraticIncreasingEscrow.sol#240)
VotingEscrow.sweep() (src/escrow/increasing/VotingEscrowIncreasing.sol#330-342) uses a dangerous strict equality:
        - excess == 0 (src/escrow/increasing/VotingEscrowIncreasing.sol#337)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in VotingEscrow.withdraw(uint256) (src/escrow/increasing/VotingEscrowIncreasing.sol#299-327):
        External calls:
        - fee = IExitQueue(queue).exit(_tokenId) (src/escrow/increasing/VotingEscrowIncreasing.sol#313)
        - IERC20Upgradeable(token).safeTransfer(address(queue),fee) (src/escrow/increasing/VotingEscrowIncreasing.sol#315)
        State variables written after the call(s):
        - _locked[_tokenId] = LockedBalance(0,0) (src/escrow/increasing/VotingEscrowIncreasing.sol#319)
        VotingEscrow._locked (src/escrow/increasing/VotingEscrowIncreasing.sol#56) can be used in cross function reentrancies:
        - VotingEscrow.locked(uint256) (src/escrow/increasing/VotingEscrowIncreasing.sol#183-185)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
QuadraticIncreasingEscrow._checkpoint(uint256,ILockedBalanceIncreasing.LockedBalance,ILockedBalanceIncreasing.LockedBalance).uNew (src/escrow/increasing/QuadraticIncreasingEscrow.sol#299) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
ERC1967Upgrade._upgradeToAndCall(address,bytes,bool) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#59-64) ignores return value by Address.functionDelegateCall(newImplementation,data) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#62)
ERC1967Upgrade._upgradeBeaconToAndCall(address,bytes,bool) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#150-156) ignores return value by Address.functionDelegateCall(IBeacon(newBeacon).implementation(),data) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#154)
ERC1967UpgradeUpgradeable._upgradeToAndCall(address,bytes,bool) (lib/openzeppelin-contracts-upgradeable/contracts/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#65-70) ignores return value by AddressUpgradeable.functionDelegateCall(newImplementation,data) (lib/openzeppelin-contracts-upgradeable/contracts/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#68)
ERC1967UpgradeUpgradeable._upgradeBeaconToAndCall(address,bytes,bool) (lib/openzeppelin-contracts-upgradeable/contracts/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#156-162) ignores return value by AddressUpgradeable.functionDelegateCall(IBeacon(newBeacon).implementation(),data) (lib/openzeppelin-contracts-upgradeable/contracts/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#160)
ProxyLib.deployMinimalProxy(address,bytes) (src/libs/ProxyLib.sol#33-41) ignores return value by minimalProxy.functionCall({data:_initCalldata}) (src/libs/ProxyLib.sol#39)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Lock.initialize(address,string,string,address)._name (src/escrow/increasing/Lock.sol#54) shadows:
        - ERC721Upgradeable._name (lib/openzeppelin-contracts-upgradeable/contracts/token/ERC721/ERC721Upgradeable.sol#25) (state variable)
Lock.initialize(address,string,string,address)._symbol (src/escrow/increasing/Lock.sol#55) shadows:
        - ERC721Upgradeable._symbol (lib/openzeppelin-contracts-upgradeable/contracts/token/ERC721/ERC721Upgradeable.sol#28) (state variable)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
ExitQueue.initialize(address,uint256,address,uint256,address,uint256)._escrow (src/escrow/increasing/ExitQueue.sol#57) lacks a zero-check on :
                - escrow = _escrow (src/escrow/increasing/ExitQueue.sol#65)
```

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.