

ABRACADABRA MONEY · SECURITY REVIEW

Abracadabra Money PurrSwap Aerodrome.

A security review of PurrSwap Aerodrome.

CLIENT	PROJECT	AUDITORS	AUDIT WINDOW	FINAL REPORT
Abracadabra Money	PurrSwap Aerodrome	0xCiphky ., Zdravko Hristov, Nicholas Chew	June 27 – August 6, 2025	August 6, 2025

§ 00

Contents.

146 pages: overview, confidence ranking, scope, 105 findings, disclaimer, and about Guardian.

§ 01	Overview	P. 03
§ 02	Confidence Ranking	P. 04
§ 03	Scope & Classifications	P. 05
	Files in Scope	P. 06
§ 04	Findings	P. 08
	Main Review · 95 findings	P. 13
	Remediation Review · 10 findings	P. 134
§ 05	Disclaimer	P. 145
§ 06	About Guardian	P. 146

§ 01

Overview.

Guardian reviewed PurrSwap Aerodrome across 2 review rounds. The final report includes 9 High, 23 Medium, 45 Low, 28 Informational findings.

AUDIT FIRM	REPOSITORY
Guardian	https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team1
CLIENT	METHODOLOGY
Abracadabra Money	Smart contract · Manual · Main Review · Remediation Review
PROJECT	MAIN REVIEW COMMIT
PurrSwap Aerodrome	ed3147a45b1b
PREPARED BY	REMEDIATION REVIEW COMMIT
OxCiphky ., Zdravko Hristov, Nicholas Chew	ecf8e47187e5
AUDIT WINDOW	
June 27 – August 6, 2025	
FINAL REPORT	
August 6, 2025	

1.1 Confidence Ranking

2 / 5

Low Confidence

Guardian assigns a Confidence Ranking of 2 out of 5, combining finding severity, remediation outcome, and the reviewer’s assessment of latent risk. The full scale is on the next page.

1	2	3	4	5
VERY LOW CONFIDENCE	LOW CONFIDENCE	MODERATE CONFIDENCE	HIGH CONFIDENCE	VERY HIGH CONFIDENCE

1.2 Findings at a Glance

SEVERITY	TOTAL	PENDING	ACK	PARTIAL	RESOLVED	FIX REVIEW
Critical	0	0	0	0	0	0
High	9	0	0	0	7	2
Medium	23	2	2	0	16	3
Low	45	8	5	0	29	3
Informational	28	26	0	0	0	2
All severities	105	36	7	0	52	10

§ 02

Confidence Ranking.

Guardian assigns a project-level confidence score using the scale below. The ranking combines finding severity, remediation outcome, and the reviewer's assessment of latent risk.

CONFIDENCE RANKING	DEFINITION AND RECOMMENDATION	RISK PROFILE
5 Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. Recommendation: code is highly secure at time of audit.	0 High/Critical findings and few Low/Medium severity findings.
4 High Confidence	Code is clean, well-structured, and adheres to best practices. Design patterns are sound and test coverage is strong. Recommendation: suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity findings.
3 Moderate Confidence	Code is functional but shows gaps in defensive programming or test coverage. Recommendation: remediate the findings and re-review the changes before deployment.	Some High/Critical findings, with various Medium severity findings.
2 Low Confidence ASSIGNED	Recurring issues suggest systemic gaps rather than isolated mistakes. Recommendation: remediate, then commission a follow-up review at a frozen commit.	Over 6 High and Critical findings across the engagement.
1 Very Low Confidence	The volume and severity of findings indicate the codebase is not ready for production. Recommendation: substantial rework, then a full re-review.	Over 12 High and Critical findings across the engagement.

§ 03

Scope & Classifications.

Guardian reviewed PurrSwap Aerodrome in

<https://github.com/Guardian0rg/purrrswap-contracts-purrrswap-aerodrome-team1>. The scope includes 6,524 source lines across 33 files.

3.1 Severity Matrix

	IMP · HIGH	IMP · MED	IMP · LOW
LIK · HIGH	Critical	High	Medium
LIK · MED	High	Medium	Low
LIK · LOW	Medium	Low	Low

IMPACT · HIGH

Significant loss of assets, harm to a group of users, or core protocol functionality is disrupted.

IMPACT · MEDIUM

Small amount of funds can be lost, or ancillary functionality is affected. Users may experience reduced or delayed receipt.

IMPACT · LOW

Unexpected behavior in some functionality that is notable but does not meet the criteria for higher severity.

LIKELIHOOD · HIGH

Possible with reasonable assumptions that mimic on-chain conditions; cost is low compared to amount gained.

LIKELIHOOD · MEDIUM

Possible only in uncommon cases or requires a large amount of capital relative to disruption caused.

LIKELIHOOD · LOW

Unlikely to ever occur in production.

Files in Scope 1/2.

FILE	NSLOC	TOTAL
MAIN REVIEW		
contracts/AirdropDistributor.sol	38	48
contracts/EpochGovernor.sol	19	34
contracts/factories/FactoryRegistry.sol	68	108
contracts/factories/GaugeFactory.sol	8	11
contracts/factories/ManagedRewardsFactory.sol	11	15
contracts/factories/VotingRewardsFactory.sol	10	14
contracts/Forwarder.sol	99	156
contracts/Gauge.sol	170	239
contracts/governance/GovernorCountingMajority.sol	54	89
contracts/governance/GovernorSimple.sol	253	472
contracts/governance/GovernorSimpleVotes.sol	28	49
contracts/governance/VetoGovernor.sol	298	530
contracts/governance/VetoGovernorCountingSimple.sol	49	91
contracts/governance/VetoGovernorVotes.sol	28	51
contracts/governance/VetoGovernorVotesQuorumFraction.sol	51	119
contracts/libraries/BalanceLogicLibrary.sol	89	132
contracts/libraries/DelegationLogicLibrary.sol	82	140
contracts/libraries/ProtocolTimeLibrary.sol	24	34
contracts/Minter.sol	148	204
contracts/ProtocolForwarder.sol	5	8
contracts/ProtocolGovernor.sol	68	91
contracts/PSPS.sol	19	29
contracts/PurrlPStakingForPoints.sol	33	43
contracts/rewards/BribeVotingReward.sol	15	23
contracts/rewards/FeesVotingReward.sol	12	19
contracts/rewards/FreeManagedReward.sol	21	33
contracts/rewards/LockedManagedReward.sol	19	32
contracts/rewards/ManagedReward.sol	15	24
contracts/rewards/Reward.sol	169	247
contracts/rewards/VotingReward.sol	22	33
contracts/RewardsDistributor.sol	148	194
contracts/Voter.sol	380	501
contracts/VotingEscrow.sol	742	1,214
REMEDIATION REVIEW		
contracts/AirdropDistributor.sol	38	48

Files in Scope 2/2.

FILE	NSLOC	TOTAL
contracts/EpochGovernor.sol	19	34
contracts/factories/FactoryRegistry.sol	68	108
contracts/factories/GaugeFactory.sol	8	11
contracts/factories/ManagedRewardsFactory.sol	11	15
contracts/factories/VotingRewardsFactory.sol	10	14
contracts/Forwarder.sol	99	156
contracts/Gauge.sol	174	246
contracts/governance/GovernorCountingMajority.sol	54	89
contracts/governance/GovernorSimple.sol	277	507
contracts/governance/GovernorSimpleVotes.sol	28	49
contracts/governance/VetoGovernor.sol	308	551
contracts/governance/VetoGovernorCountingSimple.sol	49	91
contracts/governance/VetoGovernorVotes.sol	33	59
contracts/governance/VetoGovernorVotesQuorumFraction.sol	51	119
contracts/libraries/BalanceLogicLibrary.sol	89	132
contracts/libraries/DelegationLogicLibrary.sol	82	140
contracts/libraries/ProtocolTimeLibrary.sol	24	34
contracts/Minter.sol	148	204
contracts/ProtocolForwarder.sol	5	8
contracts/ProtocolGovernor.sol	77	111
contracts/PSPS.sol	19	29
contracts/PurrlPStakingForPoints.sol	33	43
contracts/rewards/BribeVotingReward.sol	15	23
contracts/rewards/FeesVotingReward.sol	12	19
contracts/rewards/FreeManagedReward.sol	21	33
contracts/rewards/LockedManagedReward.sol	19	32
contracts/rewards/ManagedReward.sol	15	24
contracts/rewards/Reward.sol	169	247
contracts/rewards/VotingReward.sol	22	33
contracts/RewardsDistributor.sol	148	194
contracts/Voter.sol	420	572
contracts/VotingEscrow.sol	784	1,283

§ 04

Findings.

Grouped by round. Visible IDs reset by severity within each round, so a finding is named the same way in the report as it was in the channel it was written in. Status is where each finding ended up; its own page says how.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
MAIN REVIEW					
H-01	Voter.Poke Function Can Be Blocked to Lock in Permanent Voting Power	DoS	HIGH	RESOLVED	p. 13
H-02	Voter.depositManaged Function Does Not Call _reset For The Normal NFT Deposited	Logical Error	HIGH	RESOLVED	p. 15
H-03	Team Emission Calculation Error	Math	HIGH	RESOLVED	p. 18
H-04	Voter.Poke Function Can Be Blocked to Lock in Permanent Voting Power	DoS	HIGH	RESOLVED	p. 19
H-05	Voter.depositManaged Function Does Not Call _reset For The Normal NFT Deposited	Logical Error	HIGH	RESOLVED	p. 21
H-06	Team Emission Calculation Error	Math	HIGH	RESOLVED	p. 24
H-07	Deactivated Locks Remain Functional	Logical Error	HIGH	RESOLVED	p. 25
M-01	Growth Formula Rewards Early Lockers Disproportionately	Math	MEDIUM	PENDING	p. 26
M-02	Potential Loss Of A Portion Of Managed Locked Rewards	Warning	MEDIUM	RESOLVED	p. 28
M-03	Team Emission Rate Never Initialized	Configuration	MEDIUM	RESOLVED	p. 29
M-04	Users Can Make Their veNFT Unpokeable By Voting For A To-Be-Killed Gauge	Validation	MEDIUM	RESOLVED	p. 30
M-05	PurrLPStakingForPoints Stakers Lose LP Trading Fees	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 32
M-06	Users Can Make Their veNFT Unpokeable By Voting For A To-Be-Killed Gauge	Validation	MEDIUM	RESOLVED	p. 34
M-07	Bribe Rewards Distributed In Epochs With No Depositors Are Lost	Logical Error	MEDIUM	PENDING	p. 36
M-08	Potential Loss Of A Portion Of Managed Locked Rewards	Warning	MEDIUM	RESOLVED	p. 37
M-09	Calldata-Length Execution Bypass In EpochGovernor	Validation	MEDIUM	RESOLVED	p. 38
M-10	A Portion Of LP Fees Will Be Locked After A Gauge Kill	Rewards	MEDIUM	RESOLVED	p. 40

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
M-11	A Portion Of LP Fees Will Be Locked After A Gauge Kill	Rewards	MEDIUM	RESOLVED	p. 41
M-12	Wrong DOMAIN_TYPEHASH Definition In VotingEscrow Contract	Best Practices	MEDIUM	RESOLVED	p. 42
M-13	Signatures Can Be Replayed In Governor Contracts	Signatures	MEDIUM	RESOLVED	p. 44
M-14	Calldata-Length Execution Bypass In GovernorSimple	Validation	MEDIUM	RESOLVED	p. 45
M-15	Deactivated Locks Allowed In Governance Vote	Logical Error	MEDIUM	RESOLVED	p. 47
M-16	Deployment Is Missing Key Operations	Configuration	MEDIUM	RESOLVED	p. 48
M-17	PurrLPStakingForPoints Stakers Lose LP Trading Fees	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 49
M-18	Whitelist Restriction Bypassed Via poke	Access Control	MEDIUM	RESOLVED	p. 51
M-19	Incorrect Sender Handling In reset	DoS	MEDIUM	RESOLVED	p. 52
M-20	Large Voter Can Trigger Quorum At Last Minute	Logical Error	MEDIUM	RESOLVED	p. 53
L-01	Precision Loss in Slope Calculation	Math	LOW	ACKNOWLEDGED	p. 54
L-02	Lack Of Decimal-precision Check For Lock Token In VotingEscrow	Warning	LOW	ACKNOWLEDGED	p. 55
L-03	depositManaged Skips Pending Reward Claims	Warning	LOW	RESOLVED	p. 56
L-04	Flash-Vote Bypass via split()	Validation	LOW	RESOLVED	p. 57
L-05	Unclaimed Rewards Are Not Counted In getPastVotes Computation	Warning	LOW	PENDING	p. 58
L-06	Redundant Queued State Execution Gate	Validation	LOW	RESOLVED	p. 63
L-07	withdrawManaged Skips Pending Reward Claims	Warning	LOW	RESOLVED	p. 64
L-08	Managed NFTs Can Vote More Than Once Per Epoch	Warning	LOW	RESOLVED	p. 65
L-09	Bribe And Fee Token Emissions Can Be Gamed By Users	Warning	LOW	PENDING	p. 66
L-10	Wrong DOMAIN_TYPEHASH Definition In VotingEscrow Contract	Best Practices	LOW	RESOLVED	p. 67
L-11	Forwarder EIP-7702 Compatability	Warning	LOW	RESOLVED	p. 69
L-12	Killed Gauge May Accrue Rewards	Rewards	LOW	RESOLVED	p. 70
L-13	maxVotingNum can be bypassed	Validation	LOW	RESOLVED	p. 71
L-14	Gauge Rewards Are Lost When totalWeight == 0	Rewards	LOW	RESOLVED	p. 72

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L - 15	AirdropDistributor's safeTransferFrom Enables DOS	DoS	LOW	RESOLVED	p. 73
L - 16	Missing Require Check In EpochGovernor.propose Function	Validation	LOW	RESOLVED	p. 74
L - 17	Wrong Assumption In Vote.withdrawManaged()	Informational	LOW	RESOLVED	p. 75
L - 18	No Minimum Duration For Locks	MEV	LOW	RESOLVED	p. 77
L - 19	Unclaimed Rewards Become Inaccessible	DoS	LOW	RESOLVED	p. 78
L - 20	killGauge Forwards Rewards to Minter	Rewards	LOW	RESOLVED	p. 79
L - 21	Missing Require Check In EpochGovernor.propose Function	Validation	LOW	RESOLVED	p. 80
L - 22	Team Controls Proposal Threshold Without Governance	Warning	LOW	PENDING	p. 81
L - 23	Incorrect prior Results In Reward Contract	Unexpected Behavior	LOW	RESOLVED	p. 82
L - 24	Potential OOG Revert For withdrawManaged	Gas Optimization	LOW	ACKNOWLEDGED	p. 83
L - 25	Some Proposals Cannot Be Executed	Unexpected Behavior	LOW	RESOLVED	p. 84
L - 26	Flash-Vote Bypass via split() and merge()	Validation	LOW	RESOLVED	p. 85
L - 27	Unnecessary Check In GovernanceSimple	Gas Optimization	LOW	RESOLVED	p. 86
L - 28	No Flash-Loan Protection In balanceOfNFTAt	Logical Error	LOW	RESOLVED	p. 87
L - 29	Growth Formula Rewards Early Lockers Disproportionately	Math	LOW	PENDING	p. 88
L - 30	Low-Decimal Tokens Can Be Griefed	Rewards	LOW	ACKNOWLEDGED	p. 90
L - 31	Stale Proposals Can Be Revived Indefinitely	Logical Error	LOW	RESOLVED	p. 91
L - 32	Bribe Rewards Distributed In Epochs With No Depositors Are Lost	Logical Error	LOW	PENDING	p. 92
L - 33	Precision Loss Causes Irrecoverable Rewards	Rewards	LOW	ACKNOWLEDGED	p. 93
L - 34	Valid Proposals Do Not Have Deadlines	Unexpected Behavior	LOW	RESOLVED	p. 94
L - 35	Incorrect Logic In Voter/Distributor Check	Logical Error	LOW	RESOLVED	p. 95
L - 36	Expired Token Can Be Merged to Extend Lock	Logical Error	LOW	RESOLVED	p. 96
L - 37	EIP compliance issues	Unexpected Behavior	LOW	RESOLVED	p. 97
L - 38	Poke Can Be Blocked By Transfer	DoS	LOW	PENDING	p. 99
L - 39	Team Controls Proposal Threshold Without Governance	Warning	LOW	PENDING	p. 100

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-40	withdrawManaged Skips Pending Reward Claims	Warning	LOW	RESOLVED	p. 101
L-41	Zero Locked Supply Allows Bypass Of Quorum	Warning	LOW	PENDING	p. 102
L-42	Whitelisted Rewards Persist After Removal	Logical Error	LOW	RESOLVED	p. 103
I-01	Unit-mismatch In _claimFees Can Prevent Fee Distribution	Logical Error	INFORMATIONAL	PENDING	p. 104
I-02	Incorrect Time-Constant Comparison In _distribute()	Logical Error	INFORMATIONAL	PENDING	p. 106
I-03	Missing Contract-wallet Delegation Support	Warning	INFORMATIONAL	PENDING	p. 108
I-04	Unused Imports	Warning	INFORMATIONAL	PENDING	p. 109
I-05	Transient Storage Can Be Used For The ReentrancyGuard	Warning	INFORMATIONAL	PENDING	p. 110
I-06	veNFT Splitting Is Disabled By Default	Configuration	INFORMATIONAL	PENDING	p. 111
I-07	Missalignment With Velodrome Tokenomics	Warning	INFORMATIONAL	PENDING	p. 112
I-08	Unused _proposalProposer Internal View Function	Warning	INFORMATIONAL	PENDING	p. 114
I-09	Best Practices For PSPS Reward Distribution For PurrLPStakingForPoints Stakers	Warning	INFORMATIONAL	PENDING	p. 115
I-10	Accounts Can Own Multiple Managed NFTs	Informational	INFORMATIONAL	PENDING	p. 116
I-11	Deceiving Comment In split Function	Informational	INFORMATIONAL	PENDING	p. 117
I-12	Transfer to Zero Address Allowed	Logical Error	INFORMATIONAL	PENDING	p. 118
I-13	Request type is not closed properly	Best Practices	INFORMATIONAL	PENDING	p. 119
I-14	Gauges With isPool = false Never Receive Fees	Gas Optimization	INFORMATIONAL	PENDING	p. 120
I-15	transfer And unlock Constraints	Informational	INFORMATIONAL	PENDING	p. 121
I-16	lockPermanent can be applied to voting tokens	Informational	INFORMATIONAL	PENDING	p. 122
I-17	Auto-compound Set To True Disables Gauge Fee Stream	Warning	INFORMATIONAL	PENDING	p. 123
I-18	External Functions Naming	Best Practices	INFORMATIONAL	PENDING	p. 125
I-19	increaseAmount () Donates To Rewards	Informational	INFORMATIONAL	PENDING	p. 126
I-20	Immediate Execution Lacks Timelock Delay	Warning	INFORMATIONAL	PENDING	p. 127
I-21	Setters Lack Event Emissions	Best Practices	INFORMATIONAL	PENDING	p. 128
I-22	LOCKED and MANAGED Delegate Differences	Informational	INFORMATIONAL	PENDING	p. 129
I-23	Immediate Execution Lacks Timelock Delay	Warning	INFORMATIONAL	PENDING	p. 130
I-24	Unbounded Reward List In MultiRewards	DoS	INFORMATIONAL	PENDING	p. 131

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
I-25	No Quorum Requirement In EpochGovernor	Best Practices	INFORMATIONAL	PENDING	p. 132
I-26	Delayed Fee Distribution From _claimFees	Logical Error	INFORMATIONAL	PENDING	p. 133

REMEDIATION REVIEW

H-01	Disabling managed lock mishandles delegations	Logical Error	HIGH	REPORTED	p. 134
H-02	Withdrawal Blocked for Deactivated NFTs	DoS	HIGH	REPORTED	p. 135
M-01	Tail Emissions Cannot Be Maintained Or Decreased	Logical Error	MEDIUM	REPORTED	p. 136
M-02	Vote Reset Blocks Reward Claims	Rewards	MEDIUM	REPORTED	p. 137
M-03	Insufficient Approval Breaks Reward Claims	Rewards	MEDIUM	REPORTED	p. 138
L-01	Gauge Fees Cannot Be Claimed Post-Kill	Logical Error	LOW	REPORTED	p. 139
L-02	FreeManagedReward Fails To Revalidate Token Whitelist	Validation	LOW	REPORTED	p. 140
L-03	Deactivated NFTs Can Vote in EpochGovernor	Validation	LOW	REPORTED	p. 142
I-01	Full voting power redistributed to alive gauges	Informational	INFORMATIONAL	REPORTED	p. 143
I-02	Redundant Condition Check in setManagedState Function	Warning	INFORMATIONAL	REPORTED	p. 144

HIGH **H-01**

RESOLVED

Voter.Poke Function Can Be Blocked to Lock in Permanent Voting Power

CATEGORY	DoS
LOCATION	Voter.sol
REVIEW	Main Review

— DESCRIPTION

The `poke(uint256 _tokenId)` function in the `Voter` contract is intended to refresh a user's gauge-voting power based on their current veNFT balance, especially important once locks decay or expire. However, its internal logic can be permanently blocked, allowing a user to carry stale voting weight indefinitely:

1. Voting with a very high amount of tokens but for a very short lock, trying to maximize the `slope` calculation.
2. As time passes, weight decays, but unless the user calls `vote`, `reset`, or `poke`, the protocol continues using the old, higher weight.
3. Once the veNFT expires, any call to `poke` will revert here, within the internal `_vote` call: `if (_poolWeight == 0) revert ZeroBalance();`
4. This way the user has gained permanent voting power as no one can actually call `poke` to readjust it. Whenever the user wants to exit, he just has to call `reset` and `withdraw`.

By exploiting this, a user can effectively lock in permanent voting power for a veNFT without ever switching to a genuine "permanent lock" (`lockPermanent`). Their decayed lock still behaves as if it were permanent and there is no way to remove it.

****Proof Of Concept****

```
```solidity
function test_POV() public {
/*
current timestamp 1750896000 - 3612 = 1750892388
next epoch starts at 1750896000
uNew.slope = _newLocked.amount / iMAXTIME;
uNew.bias = uNew.slope * (_newLocked.end - block.timestamp).toInt128();
*/
}
```

**H-01** Voter.Poke Function Can Be Blocked to Lock in Permanent Voting Power **CONTINUED**

```
vm.startPrank(address(contract_Minter));
contract_PSPS.mint(user1, 1000000e18);
vm.stopPrank();

vm.startPrank(user1);
contract_PSPS.approve(address(contract_VotingEscrow), type(uint256).max);
uint256 nld1 = contract_VotingEscrow.createLock(1000000e18, 3612);
_consoleLogveNFTData(nld1);

address[] memory pools = new address[](1);
pools[0] = address(contract_purrLP_USDT0_USDE);
uint256[] memory weights = new uint256[](1);
weights[0] = 10000;
contract_Voter.vote(1, pools, weights);

console.log("\n3612 seconds later");
vm.warp(block.timestamp + 3612);
vm.roll(block.number + 1);
contract_Minter.updatePeriod();

console.log("\nTWO HOUR LATER"); // 1 hour of distribute window
vm.warp(block.timestamp + 2 hours);
vm.roll(block.number + (2 hours / 12 seconds));

contract_Voter.reset(nld1);

contract_VotingEscrow.withdraw(nld1);
vm.stopPrank();
}
...
```

**RECOMMENDATION**

Consider updating the `poke` function to avoid reverting when a calculated gauge weight is zero, for example, by skipping such gauges rather than failing the entire transaction.

**RESOLUTION**

Status: Resolved.

HIGH **H-02**

RESOLVED

# Voter.depositManaged Function Does Not Call `_reset` For The Normal NFT Deposited

CATEGORY Logical Error

LOCATION Voter.sol

REVIEW Main Review

## DESCRIPTION

`Voter.depositManaged` migrates a `NORMAL` veNFT into a `MANAGED` veNFT but never purges the old NFT's voting records by calling the `_reset` function. Therefore:

- When the `NORMAL` NFT had voted earlier, `weights[pool]`, `votes[_tokenId][pool]`, `usedWeights[_tokenId]` and `totalWeight` still include its weight `W`.
- After the lock is transferred, `_mTokenId` holds the real voting power and `_poke` adds another `W` to the chosen pools and to `totalWeight`.
- The system now counts  $2 \times W$  although only `W` tokens are locked.

Because the original NFT's weight is never withdrawn from the bribe and fee reward contracts, that placeholder continues to accrue its historical share of external incentives. Meanwhile the managed NFT starts earning a second share for the same underlying tokens. Both bribes and protocol-collected fees can therefore be harvested twice, one claim per token ID, until manual intervention removes the stale balance. The inconsistency remains as `poke` can not clean it up because the call reverts once the old token's weight is zero, and a `LOCKED` NFT cannot vote again to trigger a reset. Unless the owner voluntarily un-manages the NFT and calls reset, or the protocol deploys a patch, the inflated weights persist across epochs.

## Proof of Concept

```
```solidity
function test_POE() public {
    vm.startPrank(address(contract_Minter));
    contract_PSPS.mint(user1, 1337e18);
    vm.stopPrank();
    vm.startPrank(user1);
    contract_PSPS.approve(address(contract_VotingEscrow), type(uint256).max);
    uint256 id1 = contract_VotingEscrow.createLock(1337e18, 4 * 365 days);
    vm.stopPrank();
    vm.prank(allowedManager);
    uint256 mld = contract_VotingEscrow.createManagedLockFor(user1);
}
```

H-02 Voter.depositManaged Function Does Not Call _reset For The Normal NFT Deposited**CONTINUED**

```
// 1 block later
vm.warp(block.timestamp + 12);
vm.roll(block.number + 1);
vm.startPrank(user1);
address[] memory pools = new address[](1);
pools[0] = address(contract_purrLP_USDT0_USDE);
uint256[] memory weights = new uint256[](1);
weights[0] = 10000;
console.log("contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> %e",
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)));
console.log("\n FIRST VOTE WITH NORMAL NFT");
contract_Voter.vote(id1, pools, weights);
console.log("contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> %e",
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)));
// We can not deposit right away so we go into next epoch
vm.warp(block.timestamp + 7 days);
vm.roll(block.number + (7 days / 12 seconds));
contract_Minter.updatePeriod();
// Now we can
contract_Voter.depositManaged(id1, mld);
console.log("\n SECOND VOTE WITH MANAGED NFT");
contract_Voter.vote(mld, pools, weights);
uint256 currentTimestamp = block.timestamp;
console.log("contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> %e",
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)));
vm.stopPrank();
}
```

H-02 Voter.depositManaged Function Does Not Call _reset For The Normal NFT Deposited

CONTINUED

```
/*  
[PASS] test_POC() (gas: 5497094)  
Logs:  
_deployAll()  
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> 0e0
```

FIRST VOTE WITH NORMAL NFT

```
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> 1.333375142694024546e21
```

SECOND VOTE WITH MANAGED NFT

```
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> 2.670375142694024546e21
```

```
*/  
...
```

— RECOMMENDATION

Consider calling `_reset(_tokenId)` unconditionally within the `depositManaged` function.

— RESOLUTION

Status: Resolved.

HIGH **H-03**

RESOLVED

Team Emission Calculation Error

CATEGORY Math
LOCATION Minter.sol
REVIEW Main Review

DESCRIPTION

In the tail-emission branch of `updatePeriod`, the team's share is computed using the stale weekly value rather than the actual emission amount delivered to voters:

```
``solidity
bool _tail = _weekly < TAIL_START;
if (_tail) {
    _emission = (_totalSupply * tailEmissionRate) / MAX_BPS;
    // ...
    uint256 _teamEmissions = (_rate * (_growth + _weekly)) / (MAX_BPS - _rate);
}
```

Because `_weekly` remains at its pre-tail threshold value, the team's percentage share will not track the true emission volume. If actual emissions exceed `_weekly`, the team is underpaid, if emissions fall below `_weekly`, the team is overpaid. This misalignment can result in the team receiving more tokens than intended, diluting ve-locked holders and inflating total supply.

RECOMMENDATION

Calculate the team's allocation based on the actual `_emission` value (plus growth) to guarantee the correct percentage distribution each epoch.

RESOLUTION

Status: Resolved.

HIGH **H-04**

RESOLVED

Voter.Poke Function Can Be Blocked to Lock in Permanent Voting Power

CATEGORY	DoS
LOCATION	Voter.sol
REVIEW	Main Review

— DESCRIPTION

The `poke(uint256 _tokenId)` function in the `Voter` contract is intended to refresh a user's gauge-voting power based on their current veNFT balance, especially important once locks decay or expire. However, its internal logic can be permanently blocked, allowing a user to carry stale voting weight indefinitely:

1. Voting with a very high amount of tokens but for a very short lock, trying to maximize the `slope` calculation.
 2. As time passes, weight decays, but unless the user calls `vote`, `reset`, or `poke`, the protocol continues using the old, higher weight.
 3. Once the veNFT expires, any call to `poke` will revert here, within the internal `_vote` call: `if (_poolWeight == 0) revert ZeroBalance();`
 4. This way the user has gained permanent voting power as no one can actually call `poke` to readjust it. Whenever the user wants to exit, he just has to call `reset` and `withdraw`.
- By exploiting this, a user can effectively lock in permanent voting power for a veNFT without ever switching to a genuine "permanent lock" (`lockPermanent`). Their decayed lock still behaves as if it were permanent and there is no way to remove it.

****Proof Of Concept****

```

```solidity
function test_POE() public {
/*
current timestamp 1750896000 - 3612 = 1750892388
next epoch starts at 1750896000
uNew.slope = _newLocked.amount / iMAXTIME;
uNew.bias = uNew.slope * (_newLocked.end - block.timestamp).toInt128();
*/

```

**H-04** Voter.Poke Function Can Be Blocked to Lock in Permanent Voting Power **CONTINUED**

```

vm.startPrank(address(contract_Minter));
contract_PSPS.mint(user1, 1000000e18);
vm.stopPrank();

vm.startPrank(user1);
contract_PSPS.approve(address(contract_VotingEscrow), type(uint256).max);
uint256 nId1 = contract_VotingEscrow.createLock(1000000e18, 3612);
_consoleLogveNFTData(nId1);

address[] memory pools = new address[](1);
pools[0] = address(contract_purrLP_USDT0_USDE);
uint256[] memory weights = new uint256[](1);
weights[0] = 10000;
contract_Voter.vote(1, pools, weights);

console.log("\n3612 seconds later");
vm.warp(block.timestamp + 3612);
vm.roll(block.number + 1);
contract_Minter.updatePeriod();

console.log("\nTWO HOUR LATER"); // 1 hour of distribute window
vm.warp(block.timestamp + 2 hours);
vm.roll(block.number + (2 hours / 12 seconds));

contract_Voter.reset(nId1);

contract_VotingEscrow.withdraw(nId1);
vm.stopPrank();
}
` ``

```

**RECOMMENDATION**

Consider updating the `poke` function to avoid reverting when a calculated gauge weight is zero, for example, by skipping such gauges rather than failing the entire transaction.

**PROOF OF CONCEPT** <https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team1/blob/01eb1dd6376a814673bdb9c277ec7cac5ecd8cc2/test/GuardianTests/GuardianTests.t.sol#L12>

**RESOLUTION**

Status: Resolved.

HIGH **H-05**

RESOLVED

# Voter.depositManaged Function Does Not Call `_reset` For The Normal NFT Deposited

CATEGORY Logical Error

LOCATION Voter.sol

REVIEW Main Review

## — DESCRIPTION

`Voter.depositManaged` migrates a `NORMAL` veNFT into a `MANAGED` veNFT but never purges the old NFT's voting records by calling the `_reset` function. Therefore:

- When the `NORMAL` NFT had voted earlier, `weights[pool]`, `votes[_tokenId][pool]`, `usedWeights[_tokenId]` and `totalWeight` still include its weight `W`.
- After the lock is transferred, `_mTokenId` holds the real voting power and `_poke` adds another `W` to the chosen pools and to `totalWeight`.
- The system now counts  $2 \times W$  although only `W` tokens are locked.

Because the original NFT's weight is never withdrawn from the bribe and fee reward contracts, that placeholder continues to accrue its historical share of external incentives. Meanwhile the managed NFT starts earning a second share for the same underlying tokens. Both bribes and protocol-collected fees can therefore be harvested twice, one claim per token ID, until manual intervention removes the stale balance. The inconsistency remains as `poke` can not clean it up because the call reverts once the old token's weight is zero, and a `LOCKED` NFT cannot vote again to trigger a reset. Unless the owner voluntarily un-manages the NFT and calls reset, or the protocol deploys a patch, the inflated weights persist across epochs.

## Proof of Concept

```
```solidity
function test_POE() public {
    vm.startPrank(address(contract_Minter));
    contract_PSPS.mint(user1, 1337e18);
    vm.stopPrank();
    vm.startPrank(user1);
    contract_PSPS.approve(address(contract_VotingEscrow), type(uint256).max);
    uint256 id1 = contract_VotingEscrow.createLock(1337e18, 4 * 365 days);
    vm.stopPrank();
    vm.prank(allowedManager);
    uint256 mld = contract_VotingEscrow.createManagedLockFor(user1);
}
```

H-05 Voter.depositManaged Function Does Not Call _reset For The Normal NFT Deposited**CONTINUED**

```
// 1 block later
vm.warp(block.timestamp + 12);
vm.roll(block.number + 1);
vm.startPrank(user1);
address[] memory pools = new address[](1);
pools[0] = address(contract_purrLP_USDT0_USDE);
uint256[] memory weights = new uint256[](1);
weights[0] = 10000;
console.log("contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> %e",
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)));
console.log("\n FIRST VOTE WITH NORMAL NFT");
contract_Voter.vote(id1, pools, weights);
console.log("contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> %e",
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)));
// We can not deposit right away so we go into next epoch
vm.warp(block.timestamp + 7 days);
vm.roll(block.number + (7 days / 12 seconds));
contract_Minter.updatePeriod();
// Now we can
contract_Voter.depositManaged(id1, mld);
console.log("\n SECOND VOTE WITH MANAGED NFT");
contract_Voter.vote(mld, pools, weights);
uint256 currentTimestamp = block.timestamp;
console.log("contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> %e",
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)));
vm.stopPrank();
}
```

H-05 Voter.depositManaged Function Does Not Call _reset For The Normal NFT Deposited**CONTINUED**

```
/*  
[PASS] test_POC() (gas: 5497094)  
Logs:  
_deployAll()  
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> 0e0
```

FIRST VOTE WITH NORMAL NFT

```
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> 1.333375142694024546e21
```

SECOND VOTE WITH MANAGED NFT

```
contract_Voter.weights(address(contract_purrLP_USDT0_USDE)) -> 2.670375142694024546e21
```

```
*/  
...
```

— RECOMMENDATION

Consider calling `_reset(_tokenId)` unconditionally within the `depositManaged` function.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrswap-contracts-purrswap-aerodrome-team1/blob/d8c0dccc9b4ed79e34ce3c8a63b2b40279a49557/test/GuardianTests/GuardianTests.t.sol#L12>

— RESOLUTION

Status: Resolved.

HIGH **H-06**

RESOLVED

Team Emission Calculation Error

CATEGORY Math
LOCATION Minter.sol
REVIEW Main Review

DESCRIPTION

In the tail-emission branch of `updatePeriod`, the team's share is computed using the stale weekly value rather than the actual emission amount delivered to voters:

```
``solidity
bool _tail = _weekly < TAIL_START;
if (_tail) {
    _emission = (_totalSupply * tailEmissionRate) / MAX_BPS;
    // ...
    uint256 _teamEmissions = (_rate * (_growth + _weekly)) / (MAX_BPS - _rate);
}
```

Because `_weekly` remains at its pre-tail threshold value, the team's percentage share will not track the true emission volume. If actual emissions exceed `_weekly`, the team is underpaid, if emissions fall below `_weekly`, the team is overpaid. This misalignment can result in the team receiving more tokens than intended, diluting ve-locked holders and inflating total supply.

RECOMMENDATION

Calculate the team's allocation based on the actual `_emission` value (plus growth) to guarantee the correct percentage distribution each epoch.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team1/blob/b112a44a75407958919cd6efa73f36df38039f4d/test/GuardianTests/GuardianTests.t.sol#L12>

RESOLUTION

Status: Resolved.

HIGH **H-07**

RESOLVED

Deactivated Locks Remain Functional

CATEGORY	Logical Error
LOCATION	Voter.sol, VotingEscrow.sol
REVIEW	Main Review

DESCRIPTION

According to the [Velodrome specification](https://github.com/velodrome-finance/contracts/blob/main/SPECIFICATION.md#votingescrow): "Managed NFTs can be deactivated, a process which prevents the NFT from voting and from receiving deposits".

There are three problems with the way deactivated locks are handled:

- the votes they already casted stay as they are and since they are not decaying, the lock continues to vote
- `VotingEscrow.increaseAmount()` allows increasing the amount of a deactivated lock, which can be used to further increase the voting power of the NFT.
- Delegations for deactivated locks are not handled correctly because they are not reset, nor the `delegate()` function checks if the delegator is deactivated. This allows a deactivated lock to bypass the restriction and vote regardless of its status.

In result, deactivating a managed lock is insufficient action for stopping it from voting. The only limitation on it is that it cannot change the pools it votes to - they stay the same as the ones before the deactivation. Keep in mind that deactivated locks keep receiving rewards from the `RewardsDistributor` contract.

RECOMMENDATION

Consider:

- Updating `poke` in such a way that if the poked token is a deactivated managed lock, only `reset()` is executed.
- Adding a require statement in `VotingEscrow.increaseAmount()` which reverts if the managed lock is deactivated.
- Clearing the delegation once the lock is deactivated.
- Disallowing delegating from an NFT that is deactivated.

RESOLUTION

Status: Resolved.

MEDIUM **M-01**

PENDING

Growth Formula Rewards Early Lockers Disproportionately

CATEGORY Math
LOCATION Minter.sol
REVIEW Main Review

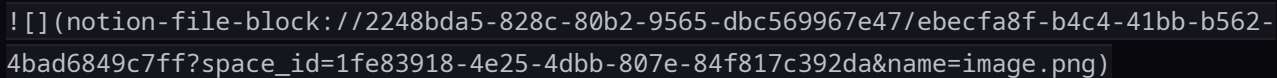
DESCRIPTION

The current `Minter.calculateGrowth` implementation completely reverses Velodrome's reward logic, giving an outsized bonus to the very first lockers:

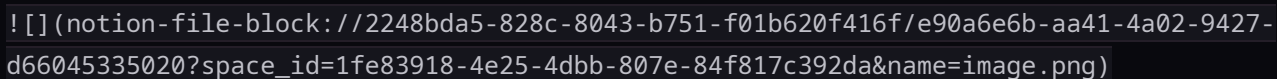
```
```solidity
// Original Velodrome: growth scales with locked share (Ve)
return ((((_minted * _veTotal) / _veloTotal) * _veTotal) / _veloTotal) * _veTotal / _veloTotal / 2;

// Current code: growth scales with circulating share ((Vtot-Ve))
return (((_minted * (_aeroTotal - _veTotal)) / _aeroTotal) * (_aeroTotal - _veTotal)) / _aeroTotal / 2;
```
```

Velodrome:



Current formula:



- Launch behaviour: With `ve.totalSupplyAt(activePeriod-1) == 0``, week 0 growth equals 50% of emissions (≈ 5 MPSPS). All of it accrues to the wallets that receive the permanent-lock NFTs immediately after, costing them nothing.

- Long-term behaviour:

- It does not incentivize locking because growth shrinks as the locked share rises, there is little incentive for the final 20–30% of holders to join the protocol once a majority has already locked.

- At $\sim 57\%$ locked, Velodrome's cubic rule would pay more growth than this quadratic-circulating rule and therefore the protection for late lockers is weaker.

- Economic impact:

- Early inflation: epoch 0 mints ≈ 15.8 M tokens instead of 10 M (+50%).

- Governance concentration: early lockers rapidly accumulate supply, keeping governance

M-01 Growth Formula Rewards Early Lockers Disproportionately

CONTINUED

control.

- APR whiplash: yields start extremely high, then decay sharply as more tokens lock.

— RECOMMENDATION

If the goal is Velodrome-style “lock-more-earn-more” incentives: revert to the cubic-locked formula. If fast boot-strapping is intentional: keep the new formula but:

- impose an upper cap on growth (e.g., $\leq X\%$ of current ve supply) to prevent extreme early windfalls.
- publicly document and justify the alternate incentive curve, and update economic models, emissions schedules and front-end displays to match the revised supply dynamics.

— RESOLUTION

Status: Pending.

MEDIUM **M-02**

RESOLVED

Potential Loss Of A Portion Of Managed Locked Rewards

| | |
|----------|--|
| CATEGORY | Warning |
| LOCATION | VotingEscrow.sol, LockedRewardsManager.sol |
| REVIEW | Main Review |

— DESCRIPTION

When a child veNFT is withdrawn from a managed lock, `VotingEscrow.withdrawManaged` harvests the child's share of locked-managed rewards by calling `lockedManagedReward.getReward(childId, [psps])` before it burns the child's weight.

The `LockedManagedReward` contract's `earned(token, childId)` function, however, only credits tokens from epochs that have already completed. Any rewards accrued during the still open epoch return zero. Because the `getReward` call also updates `lastEarn[token][childId]` to `block.timestamp`, the residual portion of the current epoch becomes permanently unclaimable: future earned calculations start counting from the moment of withdrawal, not from the epoch boundary. In effect, calling `withdrawManaged` mid-epoch irreversibly discards that week's in-progress yield for the withdrawing child, even though those tokens remain inside the managed lock and continue to influence the delegatee's voting checkpoints.

To collect every token of `LockedManagedReward` currently, users must:

- Call `RewardsDistributor.claim(managedTokenId)` at the last block of the previous epoch.
- Call `withdrawManaged` at the beginning of the new epoch.

However, if the followed the expected approach of calling `RewardsDistributor.claim(managedTokenId)` and right away call `withdrawManaged` they would lose all the just claimed yield, as that yield would only be available on the next epoch.

— RECOMMENDATION

Consider explicitly documenting this behaviour so users can withdraw following an optimal approach.

— RESOLUTION

Status: Resolved.

MEDIUM **M-03**

RESOLVED

Team Emission Rate Never Initialized

CATEGORY Configuration

LOCATION `ve.ts`

REVIEW Main Review

DESCRIPTION

The deployment script `ve.ts` completes the usual Minter initialization sequence, deploys the contract, wires in `VotingEscrow`, `Voter`, and `RewardsDistributor`, and sets the first emission period, but it never calls `Minter.setTeamRate(uint256 newRate);`

As a result the `teamRate` variable remains set to 0. During every weekly mint the protocol allocates "team share" according to that untouched default rather than the percentage defined in the published tokenomics. If the default is zero, the team receives no emissions at all; if it is a placeholder (e.g. 30bp used for tests) the team siphons an unintended fraction of every distribution. In either case, circulating-supply and treasury balances will diverge from what would be expected.

RECOMMENDATION

Amend the deployment script to invoke `Minter.setTeamRate` with the production-approved basis-point value immediately after `Minter` is initialized and before the first call to `updatePeriod`.

RESOLUTION

Status: Resolved.

MEDIUM **M-04**

RESOLVED

Users Can Make Their veNFT Unpokeable By Voting For A To-Be-Killed Gauge

CATEGORY Validation
LOCATION Voter.sol
REVIEW Main Review

DESCRIPTION

`Voter.poke` is the mechanism a third party can use to force a veNFT's vote weights to refresh when its voting power has decayed. `poke` re-casts the veNFT's existing gauge list and weights by piping them through the internal `_vote` function:

```
`solidity
function poke(uint256 tokenId) external {
    ...
    _vote(tokenId, usedWeight, poolVote[tokenId], weights[tokenId]);
}
```

Inside `_vote` every target gauge must still be marked alive:

```
`solidity
address _gauge = gauges[_pool];
if (_gauge == address(0)) revert GaugeDoesNotExist(_pool);
if (!isAlive[_gauge]) revert GaugeNotAlive(_gauge); // <-- hard revert
`
```

A malicious user can therefore make their veNFT permanently unpokeable:

1. Vote for any gauge that is likely to be "killed" soon.
2. Once owner or governance calls `killGauge(gauge)`, `isAlive[gauge]` flips to `false`.
3. Any later attempt by an external actor to call `poke(tokenId)` reverts at the `GaugeNotAlive` check, because the dead gauge is still in `poolVote[tokenId]`.
4. The original vote weights, calculated with the veNFT's old (larger) voting power, remain locked in place. Rewards and influence that should have decayed stay inflated until the NFT owner manually calls `reset`` or submits a fresh vote.

This way holders can freeze exaggerated weight indefinitely, taking extra bribes and emissions unfairly. Moreover, the protocol-level enforcement of fair decay is disabled for those positions.

M-04 Users Can Make Their veNFT Unpokeable By Voting For A To-Be-Killed Gauge CONTINUED

— RECOMMENDATION

Update the `poke` function to ignore any gauge that has `isAlive == false` instead of forwarding the call to `_vote` unchanged. A straightforward fix is to filter the stored gauge array in-memory, skipping dead gauges before invoking `_vote`.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team1/blob/2cddaafc43002363ed3141fb8c001cd8c317f3f/test/GuardianTests/GuardianTests.t.sol#L12>

— RESOLUTION

Status: Resolved.

MEDIUM **M-05**

ACKNOWLEDGED

PurrLPStakingForPoints Stakers Lose LP Trading Fees

| | |
|----------|----------------------------|
| CATEGORY | Unexpected Behavior |
| LOCATION | PurrLPStakingForPoints.sol |
| REVIEW | Main Review |

DESCRIPTION

In `PurrLPStakingForPoints` the on-chain swap fees that “belong” to liquidity providers are permanently redirected to the protocol `treasury` instead of being distributed to the stakers who “earned” them as they were the ones that provided the liquidity in the `PurrLP` pool in the first place:

```

```solidity
function claimFees() external onlyOwnerOrRoles(ROLE_REWARD_DISTRIBUTOR) {
 PurrLP lpToken = PurrLP(stakingToken);
 (uint256 claimed0, uint256 claimed1) = lpToken.claimFees();

 if (claimed0 > 0) { // BASE-token fees
 address baseToken = lpToken._BASE_TOKEN_();
 baseToken.safeTransfer(treasury, claimed0);
 }
 if (claimed1 > 0) { // QUOTE-token fees
 address quoteToken = lpToken._QUOTE_TOKEN_();
 quoteToken.safeTransfer(treasury, claimed1);
 }
 emit FeesClaimedToTreasury(claimed0, claimed1);
}
```

```

`lpToken.claimFees` returns the fee amounts owed to LP holders, tracked via `supplyIndex` and `claimable` inside each `PurrLP` clone.

Instead of keeping those tokens in the staking contract and forwarding them as rewards to the stakers, the wrapper transfers 100 % of `claimed0` and `claimed1` straight to `treasury` address.

Stakers therefore surrender all swap-fee yield and are rewarded only with the off-chain “Points” token.

M-05 PurrLPStakingForPoints Stakers Lose LP Trading Fees

CONTINUED

— RECOMMENDATION

Redesign fee harvesting so that the amounts returned by `lpToken.claimFees` are credited to stakers either by leaving them inside the staking contract for pro-rata claims or by auto-compounding into the pool. If the protocol needs a treasury cut, take a clearly specified percentage before distribution.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-06**

RESOLVED

Users Can Make Their veNFT Unpokeable By Voting For A To-Be-Killed Gauge

CATEGORY Validation
 LOCATION Voter.sol
 REVIEW Main Review

DESCRIPTION

`Voter.poke` is the mechanism a third party can use to force a veNFT's vote weights to refresh when its voting power has decayed. `poke` re-casts the veNFT's existing gauge list and weights by piping them through the internal `_vote` function:

```
``solidity
function poke(uint256 tokenId) external {
  ...
  _vote(tokenId, usedWeight, poolVote[tokenId], weights[tokenId]);
}
```

Inside `_vote` every target gauge must still be marked alive:

```
`solidity
address _gauge = gauges[_pool];
if (_gauge == address(0)) revert GaugeDoesNotExist(_pool);
if (!isAlive[_gauge]) revert GaugeNotAlive(_gauge); // <-- hard revert`
```

A malicious user can therefore make their veNFT permanently unpokeable:

1. Vote for any gauge that is likely to be "killed" soon.
2. Once owner or governance calls `killGauge(gauge)`, `isAlive[gauge]` flips to `false`.
3. Any later attempt by an external actor to call `poke(tokenId)` reverts at the `GaugeNotAlive` check, because the dead gauge is still in `poolVote[tokenId]`.
4. The original vote weights, calculated with the veNFT's old (larger) voting power, remain locked in place. Rewards and influence that should have decayed stay inflated until the NFT owner manually calls `reset`` or submits a fresh vote.

This way holders can freeze exaggerated weight indefinitely, taking extra bribes and emissions unfairly. Moreover, the protocol-level enforcement of fair decay is disabled for those positions.

M-06 Users Can Make Their veNFT Unpokeable By Voting For A To-Be-Killed Gauge **CONTINUED**

— RECOMMENDATION

Update the `poke` function to ignore any gauge that has `isAlive == false` instead of forwarding the call to `_vote` unchanged. A straightforward fix is to filter the stored gauge array in-memory, skipping dead gauges before invoking `_vote`.

— RESOLUTION

Status: Resolved.

MEDIUM **M-07**

PENDING

Bribe Rewards Distributed In Epochs With No Depositors Are Lost

CATEGORY Logical Error
LOCATION BribeVotingReward.sol
REVIEW Main Review

— DESCRIPTION

Rewards that are sent to `BribeVotingReward` vaults are lost if there are no vault depositors for that epoch. Anyone can send bribe voting rewards to vaults by calling `BribeVotingReward.notifyRewardAmount`. This updates the mapping `tokenRewardsPerEpoch[token][epochStart]` which stores the token rewards per epoch with the newly added amount. Since the rewards are stored and distributed in their entirety by epochs, they will be lost and irrecoverable if there are no vault depositors for those epochs.

— RECOMMENDATION

Consider implementing a new function `renotifyRewardAmount` allowing for undistributed rewards to be redistributed.

— RESOLUTION

Status: Pending.

MEDIUM **M-08**

RESOLVED

Potential Loss Of A Portion Of Managed Locked Rewards

| | |
|----------|--|
| CATEGORY | Warning |
| LOCATION | VotingEscrow.sol, LockedRewardsManager.sol |
| REVIEW | Main Review |

— DESCRIPTION

When a child veNFT is withdrawn from a managed lock, `VotingEscrow.withdrawManaged` harvests the child's share of locked-managed rewards by calling `lockedManagedReward.getReward(childId, [psps])` before it burns the child's weight.

The `LockedManagedReward` contract's `earned(token, childId)` function, however, only credits tokens from epochs that have already completed. Any rewards accrued during the still open epoch return zero. Because the `getReward` call also updates `lastEarn[token][childId]` to `block.timestamp`, the residual portion of the current epoch becomes permanently unclaimable: future earned calculations start counting from the moment of withdrawal, not from the epoch boundary. In effect, calling `withdrawManaged` mid-epoch irreversibly discards that week's in-progress yield for the withdrawing child, even though those tokens remain inside the managed lock and continue to influence the delegatee's voting checkpoints.

To collect every token of `LockedManagedReward` currently, users must:

- Call `RewardsDistributor.claim(managedTokenId)` at the last block of the previous epoch.
- Call `withdrawManaged` at the beginning of the new epoch.

However, if they followed the expected approach of calling `RewardsDistributor.claim(managedTokenId)` and right away call `withdrawManaged` they would lose all the just claimed yield, as that yield would only be available on the next epoch.

— RECOMMENDATION

Consider explicitly documenting this behaviour so users can withdraw following an optimal approach.

— RESOLUTION

Status: Resolved.

MEDIUM **M-09**

RESOLVED

Calldata-Length Execution Bypass In EpochGovernor

CATEGORY Validation
LOCATION EpochGovernor.sol
REVIEW Main Review

DESCRIPTION

The `GovernorSimple.execute` function currently authorizes execution of proposals in the `Defeated` or `Expired` state, and `propose` only verifies that `bytes4(calldatas[0]) == IMinter.nudge.selector` without checking the full length of `calldatas[0]`. As a result, an attacker can submit a proposal whose calldata is the 4-byte `nudge` selector plus any extra bytes (enough to cause the `hashProposal` to return a total different new hash during the `propose` call). Because the EVM dispatch mechanism ignores trailing data when routing to a no-argument function, the `execute` call succeeds, even for a defeated proposal, and invokes `nudge` with the extra payload. Inside `nudge`, a defeated result triggers a decrement of `tailEmissionRate`, so an attacker with minimal voting weight can force repeated downward adjustments regardless of genuine community votes. For reference:

```
``solidity
// GovernorSimple.sol
require(
    status == ProposalState.Succeeded ||
    status == ProposalState.Defeated ||
    status == ProposalState.Expired,
    "GovernorSimple: proposal not successful"
);
// ...
(bool success, ) = targets[0].call{ value: values[0] }(calldatas[0]);
Address.verifyCallResult(success, returndata);
`;
```

Because `nudge`` does not revert on extra calldata, this gap lets anyone bypass the intended governance checks and manipulate protocol parameters each epoch.

RECOMMENDATION

Restrict `execute` to only allow `ProposalState.Succeeded`, and in `propose` enforce that `calldatas[0]` is exactly 4 bytes long by adding ``require(bytes(calldatas[0]).length ==`

M-09 Calldata-Length Execution Bypass In EpochGovernor

CONTINUED

4) . This ensures that only properly formed, successful proposals can invoke `nudge`` and prevents abuse via trailing-data payloads.

— RESOLUTION

Status: Resolved.

MEDIUM **M-10**

RESOLVED

A Portion Of LP Fees Will Be Locked After A Gauge Kill

CATEGORY Rewards
LOCATION Gauge.sol
REVIEW Main Review

— DESCRIPTION

In the PurrLP-based `Gauge` contract, the `_claimFees` function is only invoked via `notifyRewardAmount` (called by the `Voter`). When a gauge is “killed” in the `Voter` contract, no `notifyRewardAmount` is run beforehand, so any fees accrued in the underlying `PurrLP` contract never flow into the gauge’s fee-reward pool. Those unclaimed fees remain stranded in the `PurrLP` contract. Additionally, killed gauges may still hold user deposits and continue accruing new fees, but because the gauge is inactive, `_claimFees` cannot be triggered and those ongoing fees also become permanently inaccessible.

— RECOMMENDATION

Before marking a gauge as killed, the Voter should invoke `_claimFees` to sweep any outstanding PurrLP fees into the gauge’s reward pool. Additionally, consider unlocking or exposing a post-kill fee-claim path so that any future fees accrued to a deactivated gauge can still be claimed by the protocol or stakers.

— RESOLUTION

Status: Resolved.

MEDIUM **M-11**

RESOLVED

A Portion Of LP Fees Will Be Locked After A Gauge Kill

CATEGORY Rewards
LOCATION Gauge.sol
REVIEW Main Review

— DESCRIPTION

In the PurrLP-based `Gauge` contract, the `_claimFees` function is only invoked via `notifyRewardAmount` (called by the `Voter`). When a gauge is “killed” in the `Voter` contract, no `notifyRewardAmount` is run beforehand, so any fees accrued in the underlying `PurrLP` contract never flow into the gauge’s fee-reward pool. Those unclaimed fees remain stranded in the `PurrLP` contract. Additionally, killed gauges may still hold user deposits and continue accruing new fees, but because the gauge is inactive, `_claimFees` cannot be triggered and those ongoing fees also become permanently inaccessible.

— RECOMMENDATION

Before marking a gauge as killed, the Voter should invoke `_claimFees` to sweep any outstanding PurrLP fees into the gauge’s reward pool. Additionally, consider unlocking or exposing a post-kill fee-claim path so that any future fees accrued to a deactivated gauge can still be claimed by the protocol or stakers.

— RESOLUTION

Status: Resolved.

MEDIUM **M-12**

RESOLVED

Wrong DOMAIN_TYPEHASH Definition In VotingEscrow Contract

CATEGORY Best Practices
LOCATION VotingEscrow.sol
REVIEW Main Review

DESCRIPTION

The `delegateBySig` function implemented in the `VotingEscrow` contract is intended to allow off-chain signature delegation using the EIP-712, but the contract's hard-coded `DOMAIN_TYPEHASH` does not match the fields actually encoded in the domain separator. The code defines:

```
``solidity
bytes32 public constant DOMAIN_TYPEHASH =
keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");
`
a type hash for three fields ( name , chainId , verifyingContract ), yet in delegateBySig it
does:
`solidity
bytes32 domainSeparator = keccak256(
abi.encode(
DOMAIN_TYPEHASH,
keccak256(bytes(name)),
keccak256(bytes(version)),
block.chainid,
address(this)
)
);
`
```

which encodes four parameters (name , version , chainId , verifyingContract). Because the type hash and the actual ABI-encoded layout diverge, any signature produced off-chain with the correct domain (including version) will compute a different digest than the contract expects, causing every call to revert with "invalid signature." This breaks all signature-based delegation and prevents any off-chain wallet or service from delegating veNFT votes via signature.

M-12 Wrong DOMAIN_TYPEHASH Definition In VotingEscrow Contract

CONTINUED

RECOMMENDATION

According to the [EIP 712](https://eips.ethereum.org/EIPS/eip-712), in the [definition of domainSeparator](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md#definition-of-domainseparator), `string version` represents the current major version of the signing domain. Signatures from different versions should never be compatible. Consider updating the `DOMAIN_TYPEHASH` to include the `version` field:

```
``solidity
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,string
version,uint256 chainId,address verifyingContract)");
``
```

RESOLUTION

Status: Resolved.

MEDIUM **M-13**

RESOLVED

Signatures Can Be Replayed In Governor Contracts

CATEGORY Signatures

LOCATION GovernorSimple.sol: 479, VetoGovernor.sol: 528

REVIEW Main Review

DESCRIPTION

The `castVoteBySig` and `castVoteWithReasonAndParamsBySig` functions allow users to delegate voting via EIP-712 signatures. However, these signatures only bind the `proposalId`, `support`, `reason` and `params` values:

```
``plain text
_hashTypedDataV4(
keccak256(abi.encode(EXTENDED_BALLOT_TYPEHASH, proposalId, support, keccak256(bytes(reason)),
keccak256(params)))
`
```

The `tokenId` which identifies the specific NFT used for voting is not included in the signed message. As a result, if a user owns multiple voting NFTs, an attacker can obtain a valid signature from the user for one token, but reuse that same signature to cast votes for all other NFTs owned by the user.

Additionally, as a safeguard, a nonce should be added to the signature to prevent using the signature more than once for the same `tokenId`.

RECOMMENDATION

Include `tokenId` and a `nonce` in the EIP-712 signing schema to ensure that the signature is bound to a specific token and usable only once.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrswap-contracts-purrswap-aerodrome-team2/pull/6>

RESOLUTION

Status: Resolved.

MEDIUM **M-14**

RESOLVED

Calldata-Length Execution Bypass In GovernorSimple

CATEGORY Validation
LOCATION GovernorSimple.sol
REVIEW Main Review

DESCRIPTION

The `GovernorSimple.execute` function currently authorizes execution of proposals in the `Defeated` or `Expired` state, and `propose` only verifies that `bytes4(calldatas[0]) == IMinter.nudge.selector` without checking the full length of `calldatas[0]`. As a result, an attacker can submit a proposal whose calldata is the 4-byte `nudge` selector plus any extra bytes (enough to cause the `hashProposal` to return a total different new hash during the `propose` call). Because the EVM dispatch mechanism ignores trailing data when routing to a no-argument function, the `execute` call succeeds, even for a defeated proposal, and invokes `nudge` with the extra payload. Inside `nudge`, a defeated result triggers a decrement of `tailEmissionRate`, so an attacker with minimal voting weight can force repeated downward adjustments regardless of genuine community votes. For reference:

```
``solidity
// GovernorSimple.sol
require(
    status == ProposalState.Succeeded ||
    status == ProposalState.Defeated ||
    status == ProposalState.Expired,
    "GovernorSimple: proposal not successful"
);
// ...
(bool success, ) = targets[0].call{ value: values[0] }(calldatas[0]);
Address.verifyCallResult(success, returndata);
`
```

Because `nudge`` does not revert on extra calldata, this gap lets anyone bypass the intended governance checks and manipulate protocol parameters each epoch.

RECOMMENDATION

Restrict `execute` to only allow `ProposalState.Succeeded`, and in `propose` enforce that `calldatas[0]` is exactly 4 bytes long by adding ``require(bytes(calldatas[0]).length ==`

M-14 Calldata-Length Execution Bypass In GovernorSimple

CONTINUED

4) . This ensures that only properly formed, successful proposals can invoke `nudge`` and prevents abuse via trailing-data payloads.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrswap-contracts-purrswap-aerodrome-team2/pull/7>

— RESOLUTION

Status: Resolved.

MEDIUM **M-15**

RESOLVED

Deactivated Locks Allowed In Governance Vote

CATEGORY Logical Error
LOCATION VetoGovernorVotes.sol: 48
REVIEW Main Review

— DESCRIPTION

In the current implementation of ProtocolGovernor, managed veNFTs that have been deactivated are still able to create and vote on proposals. This undermines the very purpose of deactivation. For example, a managed veNFT could be deactivated through a proposal due to suspected manipulation or abuse. However, since the deactivation does not restrict governance participation, the same veNFT can immediately create a new proposal to reinstate itself. If successful, it would then regain access to vote on gauge emissions and redirect rewards, effectively reversing the governance decision.

— RECOMMENDATION

Consider disallowing deactivated managed veNFTs from voting in governance. Note however that this would only check presently deactivated veNFTs and would not consider past deactivations.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team2/pull/8/files>

— RESOLUTION

Status: Resolved.

MEDIUM **M-16**

RESOLVED

Deployment Is Missing Key Operations

CATEGORY Configuration

LOCATION ve.ts

REVIEW Main Review

DESCRIPTION

The deployment script `ve.ts` completes the usual Minter initialization sequence, deploys the contract, wires in `VotingEscrow`, `Voter`, and `RewardsDistributor`, and sets the first emission period, but it never calls `Minter.setTeamRate(uint256 newRate);`

As a result the `teamRate` variable remains set to 0. During every weekly mint the protocol allocates “team share” according to that untouched default rather than the percentage defined in the published tokenomics. If the default is zero, the team receives no emissions at all; if it is a placeholder (e.g. 30bp used for tests) the team siphons an unintended fraction of every distribution. In either case, circulating supply and treasury balances will diverge from what would be expected.

On the other hand, the `setArtProxy` function is never called in the `VotingEscrow` contract. The `artProxy` address is used in the `tokenURI` function, meaning any third parties—such as marketplaces—that rely on token metadata will receive no data.

Finally, the script does not deploy either the `AirdropDistributor` or `PurrLPStakingForPoints` contracts.

RECOMMENDATION

Amend the deployment script to invoke `Minter.setTeamRate` with the production-approved basis-point value immediately after `Minter` is initialized and before the first call to `updatePeriod`.

Consider also updating the deploy script to call `setArtProxy` and also deploy the missing contracts (`AirdropDistributor` and `PurrLPStakingForPoints`) if they are intended to be included in the initial setup.

RESOLUTION

Status: Resolved.

MEDIUM **M-17**

ACKNOWLEDGED

PurrLPStakingForPoints Stakers Lose LP Trading Fees

| | |
|----------|----------------------------|
| CATEGORY | Unexpected Behavior |
| LOCATION | PurrLPStakingForPoints.sol |
| REVIEW | Main Review |

DESCRIPTION

In `PurrLPStakingForPoints` the on-chain swap fees that “belong” to liquidity providers are permanently redirected to the protocol `treasury` instead of being distributed to the stakers who “earned” them as they were the ones that provided the liquidity in the `PurrLP` pool in the first place:

```

```solidity
function claimFees() external onlyOwnerOrRoles(ROLE_REWARD_DISTRIBUTOR) {
 PurrLP lpToken = PurrLP(stakingToken);
 (uint256 claimed0, uint256 claimed1) = lpToken.claimFees();

 if (claimed0 > 0) { // BASE-token fees
 address baseToken = lpToken._BASE_TOKEN_();
 baseToken.safeTransfer(treasury, claimed0);
 }
 if (claimed1 > 0) { // QUOTE-token fees
 address quoteToken = lpToken._QUOTE_TOKEN_();
 quoteToken.safeTransfer(treasury, claimed1);
 }
 emit FeesClaimedToTreasury(claimed0, claimed1);
}
```

```

`lpToken.claimFees` returns the fee amounts owed to LP holders, tracked via `supplyIndex` and `claimable` inside each `PurrLP` clone.

Instead of keeping those tokens in the staking contract and forwarding them as rewards to the stakers, the wrapper transfers 100 % of `claimed0` and `claimed1` straight to `treasury` address.

Stakers therefore surrender all swap-fee yield and are rewarded only with the off-chain “Points” token.

M-17 PurrLPStakingForPoints Stakers Lose LP Trading Fees

CONTINUED

— RECOMMENDATION

Redesign fee harvesting so that the amounts returned by `lpToken.claimFees` are credited to stakers either by leaving them inside the staking contract for pro-rata claims or by auto-compounding into the pool. If the protocol needs a treasury cut, take a clearly specified percentage before distribution.

— RESOLUTION

Status: Acknowledged.

MEDIUM **M-18**

RESOLVED

Whitelist Restriction Bypassed Via poke

CATEGORY Access Control

LOCATION Voter.sol: 201

REVIEW Main Review

DESCRIPTION

The `Voter` contract enforces a voting window by restricting non-whitelisted tokens from casting votes in the final hour of the epoch. This is implemented in the `vote` function, which checks whether a token is whitelisted before allowing votes during the restricted period.

However, this restriction is not enforced in the `poke` function. Since `poke` can update a user's voting weight and recast their votes, users can bypass the intended restriction by first voting during the allowed window, then increasing their weight via `depositFor` or `increaseAmount`, and calling `poke` during the restricted period. This enables them to amplify their influence after the voting window has closed—without needing to be whitelisted.

Additionally, users can escalate the impact of this bypass by distributing minimal different votes across multiple tokens, then selecting a specific token to boost and call `poke` on during the restricted window. Using `merge` and `split` (if permitted), they can efficiently repeat this behaviour, allowing them to manipulate vote outcomes and influence their reward allocations outside the intended voting window.

RECOMMENDATION

Consider enforcing the same whitelist and time-based restrictions in the `poke` function to maintain consistent voting rules and prevent bypassing of the intended voting window.

RESOLUTION

Status: Resolved.

MEDIUM **M-19**

RESOLVED

Incorrect Sender Handling In reset

CATEGORY DoS
LOCATION Voter.sol: 170
REVIEW Main Review

DESCRIPTION

The `Voter` contract inherits from `ERC2771Context` to support meta-transactions, correctly using `_msgSender()` in place of `msg.sender` throughout the contract. However, the `reset` function is an exception—it directly checks `msg.sender` when verifying whether the caller is the token owner or an approved address.

As a result, if the function is called via a trusted forwarder, `msg.sender` will resolve to the forwarder contract rather than the actual user. This causes the ownership check to fail, even if the user is properly authorized, effectively breaking support for meta-transactions on this function.

RECOMMENDATION

Update the `reset` function to use `_msgSender()` instead of `msg.sender` to ensure consistent support for trusted forwarders and meta-transactions across the contract.

RESOLUTION

Status: Resolved.

MEDIUM **M-20**

RESOLVED

Large Voter Can Trigger Quorum At Last Minute

CATEGORY Logical Error
LOCATION ProtocolGovernor.sol
REVIEW Main Review

— DESCRIPTION

Currently, there is no minimum voting period after a quorum is reached. This allows a large voter to sway a vote by triggering quorum at the last minute, before other voters have any time to react to try and oppose the decision.

If a vote causes a quorum to be reached, some minimum time till execution should be enforced to prevent this.

— RECOMMENDATION

Consider adopting Velodrome's `VetoGovernorPreventLateQuorum` (link)\<https://github.com/velodrome-finance/contracts/blob/main/contracts/governance/VetoGovernorPreventLateQuorum.sol> which extends the voting period if a vote causes quorum to be reached.

— RESOLUTION

Status: Resolved.

LOW **L-01**

ACKNOWLEDGED

Precision Loss in Slope Calculation

CATEGORY Math
LOCATION VotingEscrow.sol
REVIEW Main Review

DESCRIPTION

`VotingEscrow._checkpoint` derives the decay rate of every lock with two integer divisions:

```
``solidity
// iMAXTIME = 126144000
uOld.slope = _oldLocked.amount / iMAXTIME;
...
uNew.slope = _newLocked.amount / iMAXTIME;
`[
```

Because both operands are plain integers, Solidity performs a floor-division that discards the remainder. If a user locks an amount that is not an exact multiple of `iMAXTIME`, the truncated remainder is permanently lost from the slope.

The resulting error propagates to

```
`solidity
uX.bias = uX.slope * (lockEnd - now);
`[
```

so the longer the lock, the larger the `bias` under-statement. For very small locks (`amount < iMAXTIME`) the computed slope becomes `**zero**`, meaning the NFT carries no voting power even though it has a positive balance. The inaccuracy is systematic (always rounding down) and therefore disadvantages small participants and marginally reduces total supply which in turn slightly over-states relative power for larger locks.

RECOMMENDATION

Merely informative. A possible solution would be to store slope with full fixed-point precision multiplying `amount` by a scalar (e.g. `1e18`) before dividing.

RESOLUTION

Status: Acknowledged.

LOW **L-02**

ACKNOWLEDGED

Lack Of Decimal-precision Check For Lock Token In VotingEscrow

| | |
|----------|------------------|
| CATEGORY | Warning |
| LOCATION | VotingEscrow.sol |
| REVIEW | Main Review |

— DESCRIPTION

The entire voting-power model assumes the escrowed token uses 18 decimals, yet the contract never enforces that condition. Bias and slope are computed as

```
``solidity
slope = amount / iMAXTIME; // iMAXTIME = 4 years in seconds
bias = slope * (lockEnd - now);
`
```

If the underlying token were deployed with 6 decimals, a common stable-coin format, each “whole” token represents 10^{12} wei fewer units. All balances, slopes and biases become $1e12$ times smaller than intended. A 100 USDC lock would be treated as $100 \cdot 10^6$, divided by $iMAXTIME$, and truncated to zero. Therefore, the NFT would have no voting power, would not decay and growth/reward calculations that depend on global supply would break.

— RECOMMENDATION

During construction assert `IERC20Metadata(_token).decimals() == 18` (or another constant your maths expects).

Alternatively convert all arithmetic to a fixed $1e18$ internal unit by multiplying amount by $10^{(18-\text{decimals})}$ on deposit and dividing on withdrawal, so the contract remains agnostic to token precision.

— RESOLUTION

Status: Acknowledged.

LOW **L-03**

RESOLVED

depositManaged Skips Pending Reward Claims

CATEGORY Warning
LOCATION VotingEscrow.sol
REVIEW Main Review

— DESCRIPTION

When you call `depositManaged(tokenId, mTokenId)` the contract tears down the user's original veNFT by zeroing out its lock and checkpoint, then credits that weight to the new managed NFT. However, it never claims any bribe or fee rewards that had accumulated on the original lock. The code unconditionally does:

```
``solidity
_checkpoint(tokenId, _locked[tokenId], LockedBalance(0, 0, false));
_locked[tokenId] = LockedBalance(0, 0, false);
`
```

and then moves straight on to minting into the managed lock. Any unclaimed rewards in the old bribe or fee reward contracts remain tied to the now-empty `tokenId` and become effectively stranded, because a zero-balance NFT can no longer earn or claim. Users who forget to pre-claim before calling `depositManaged` will permanently forfeit those tokens.

— RECOMMENDATION

Modify `depositManaged` so that it automatically harvests all pending rewards for the original `tokenId` before zeroing its lock. In practice this means invoking each associated `IReward.getReward(tokenId, tokens[])` (for both `managedToFees` and `managedToBribe`) or using their withdrawal hooks to sweep any accrued fees, bribes, or emissions into the user's wallet prior to moving the lock.

— RESOLUTION

Status: Resolved.

LOW **L-04**

RESOLVED

Flash-Vote Bypass via split()

| | |
|----------|------------------|
| CATEGORY | Validation |
| LOCATION | VotingEscrow.sol |
| REVIEW | Main Review |

— DESCRIPTION

The `VotingEscrow.split` function mints two brand-new veNFTs in the same block, each inheriting voting power from the original. Because these new token IDs have never been transferred, their `ownershipChange` remains zero, meaning the flash-transfer protection (which zeroes out voting power for any NFT that changed hands in the current block) does not apply. Downstream voting logic (`Voter.vote` → `balanceOfNFTAt`) uses the raw checkpoint function, so a user can:

1. Acquire veNFT #1.
2. In the same block, call `split(1, fullAmount)`.
3. Immediately call `Voter.vote` with one of the fresh IDs (#2 or #3), capturing full voting power.

— RECOMMENDATION

Consider updating the `split` function so after minting the child NFTs, set `ownershipChange[newId] = block.number` for each new token.

— RESOLUTION

Status: Resolved.

LOW L-05

PENDING

Unclaimed Rewards Are Not Counted In `getPastVotes` Computation

CATEGORY Warning

LOCATION `GovernorSimpleVotes.sol`, `VetoGovernorVotes.sol`

REVIEW Main Review

DESCRIPTION

The current `ProtocolGovernor` / `GovernorSimpleVotes` / `VetoGovernorVotes` implementation derives voting power exclusively from the raw veNFT checkpoints returned by `token.getPastVotes(account, tokenId, timepoint)`. That helper reports only the veNFT's locked balance at the requested timestamp. It does not include any rewards that have already been accrued to the veNFT but have not yet been claimed.

In this system there are three escrow types:

- NORMAL: A user-owned time lock created with `createLock`
- MANAGED: The "vault" NFT that aggregates locks deposited by `depositManaged`
- LOCKED: The placeholder left behind after a normal NFT is deposited

Both NORMAL and MANAGED veNFTs can earn two classes of yield:

1. Locked-managed rewards (rebases and compounded yield) that automatically flow into a managed NFT's balance over time.
2. Free rewards (bribes, trading fees, external incentives) that accumulate inside a reward contract and are released only when the owner calls `getReward`.

In every case the reward tokens are owed to the veNFT from the moment the reward period begins, even though the owner may choose to `claim` them later. Governance power is therefore intended to reflect the total claimable balance, locked principal plus earned but unwithdrawn rewards, at the instant a proposal's snapshot is taken.

Because `GovernorSimpleVotes` / `VetoGovernorVotes` ignore these unclaimed rewards, the snapshot understates true economic stake for both types of voting NFTs:

- For a NORMAL veNFT, any rebases accrued via the protocol's rewards distributor between the last claim and the proposal's snapshot are simply invisible.
- For a MANAGED veNFT, the problem is more severe. A managed lock's entire purpose is to compound yield and distribute it pro rata to its depositors, yet the Governor counts only the initial deposit weight, not the accumulated locked yield, unless and until someone explicitly calls `claim`. The result is that managed voters wield far less influence than their actual locked value warrants, distorting voting outcomes in favour of passive lockers and against active liquidity managers.
- If delegation is involved, the discrepancy widens further. The delegator's checkpoint

L-05 Unclaimed Rewards Are Not Counted In `getPastVotes` Computation

CONTINUED

reflects only the moment of delegation, any subsequent yield remains invisible to the delegatee's vote weight even though it is contractually theirs.

A malicious actor can exploit this gap. By timing proposals to land immediately after a large reward epoch starts, but before voters bother to claim, an attacker can submit and pass a proposal while honest holders' unclaimed rewards remain uncounted, thereby suppressing their effective voting power. Conversely, an attacker can accumulate yield on their own veNFT, claim it just before the snapshot, and enjoy a sudden one-sided spike in recorded votes that other participants cannot replicate without advance knowledge of the snapshot block.

RECOMMENDATION

Replicate the original Velodrome logic in the Governor's `_getVotes` function so that the snapshot captures the full economic weight of a veNFT at the timepoint:

```
```solidity
/**
 * Read the voting weight from the token's built in snapshot mechanism (see {Governor-_getVotes}).
 */
function _getVotes(
 address account,
 uint256 tokenId,
 uint256 timepoint,
 bytes memory /*params*/
) internal view virtual override returns (uint256) {
 IVotingEscrow.EscrowType escrowType = ve.escrowType(tokenId);
 require(escrowType != IVotingEscrow.EscrowType.MANAGED, "Governor: managed nft cannot vote");

 if (escrowType == IVotingEscrow.EscrowType.NORMAL) {
 return token.getPastVotes(account, tokenId, timepoint);
 }

 // only allow locked veNFT voting if underlying nft not delegating at timepoint
 uint256 mTokenId = ve.idToManaged(tokenId);
 uint48 index = ve.getPastCheckpointIndex(mTokenId, timepoint);
 uint256 delegatee = ve.checkpoints(mTokenId, index).delegatee;
 if (delegatee == 0) {
 // if mveNFT not delegating, voting balance = delegated balance +
 // initial contribution to mveNFT + accrued locked rewards
 uint256 delegatedBalance = token.getPastVotes(account, tokenId, timepoint);
```

**L-05** Unclaimed Rewards Are Not Counted In getPastVotes Computation**CONTINUED**

```
uint256 weight = ve.weights(tokenId, mTokenId); // initial deposit weight
uint256 _earned = ve.earned(mTokenId, tokenId, timepoint); // accrued rewards
return weight + _earned + delegatedBalance;
}

// nft locked and underlying nft delegating
// balance will only be delegated balance
return token.getPastVotes(account, tokenId, timepoint);
}

// ----- DelegationHelperLibrary.sol -----
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.19;

import {IVotingEscrow} from "../interfaces/IVotingEscrow.sol";
import {IReward} from "contracts/interfaces/IReward.sol";
import {VelodromeTimeLibrary} from "contracts/libraries/VelodromeTimeLibrary.sol";
import {SafeCastLibrary} from "./SafeCastLibrary.sol";
import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

library DelegationHelperLibrary {
 uint256 public constant DURATION = 7 days;

 /// Helper function to fetch the checkpoint for the last voting checkpoint prior to a timepoint
 /// Adapted from DelegationLogicLibrary.sol:getPastVotesIndex(uint256 tokenId, uint256 timestamp)
 function getPastCheckpointIndex(
 IVotingEscrow ve,
 uint256 mTokenId,
 uint256 timepoint
) internal view returns (uint48) {
 uint48 nCheckpoints = ve.numCheckpoints(mTokenId);
 if (nCheckpoints == 0) return 0;
 // First check most recent balance
 if (ve.checkpoints(mTokenId, nCheckpoints - 1).fromTimestamp <= timepoint) return (nCheckpoints - 1);
 // Next check implicit zero balance
 if (ve.checkpoints(mTokenId, 0).fromTimestamp > timepoint) return 0;
 }
}
```

**L-05** Unclaimed Rewards Are Not Counted In getPastVotes Computation**CONTINUED**

```
uint48 lower = 0;
uint48 upper = nCheckpoints - 1;
IVotingEscrow.Checkpoint memory cp;
while (upper > lower) {
 uint48 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
 cp = ve.checkpoints(mTokenId, center);
 if (cp.fromTimestamp == timepoint) {
 return center;
 } else if (cp.fromTimestamp < timepoint) {
 lower = center;
 } else {
 upper = center - 1;
 }
}
return lower;
}
```

/// Helper function to calculate the locked balance owed to a locked nft at a certain timepoint  
/// It calculates the rewards until the end of the current epoch. As rewards are lagged by one epoch  
/// This means it includes rewards that the user is projected to get in the following epoch.  
/// These rewards are not immediately claimable, but the user can vote as if they claimed it.  
/// Adapted from Reward.sol:earned(address token, uint256 tokenId)

```
function earned(
 IVotingEscrow ve,
 uint256 mTokenId,
 uint256 tokenId,
 uint256 timepoint
) internal view returns (uint256) {
 IReward lmr = IReward(ve.managedToLocked(mTokenId));
 if (lmr.numCheckpoints(tokenId) == 0) {
 return 0;
 }

 address _rewardToken = ve.token();
 uint256 reward = 0;
 uint256 _supply = 1;
```

**L-05** Unclaimed Rewards Are Not Counted In getPastVotes Computation**CONTINUED**

```
uint256 _currTs = VelodromeTimeLibrary.epochStart(lmr.lastEarn(_rewardToken, tokenId)); // take epoch last
claimed in as starting point
uint256 _index = lmr.getPriorBalanceIndex(tokenId, _currTs);
(uint256 _cpTs, uint256 _cpBalanceOf) = lmr.checkpoints(tokenId, _index);

// accounts for case where lastEarn is before first checkpoint
_currTs = Math.max(_currTs, VelodromeTimeLibrary.epochStart(_cpTs));

// get epochs between end of the current epoch and first checkpoint in same epoch as last claim
uint256 numEpochs = (VelodromeTimeLibrary.epochNext(timepoint) - _currTs) / DURATION;
uint256 _priorSupply;

if (numEpochs > 0) {
 for (uint256 i = 0; i < numEpochs; i++) {
 // get index of last checkpoint in this epoch
 _index = lmr.getPriorBalanceIndex(tokenId, _currTs + DURATION - 1);
 // get checkpoint in this epoch
 (_cpTs, _cpBalanceOf) = lmr.checkpoints(tokenId, _index);
 // get supply of last checkpoint in this epoch
 (_priorSupply) = lmr.supplyCheckpoints(lmr.getPriorSupplyIndex(_currTs + DURATION - 1));
 _supply = Math.max(_priorSupply, 1);
 reward += (_cpBalanceOf * lmr.tokenRewardsPerEpoch(_rewardToken, _currTs)) / _supply;
 _currTs += DURATION;
 }
}

return reward;
}
}
...
```

**— RESOLUTION**

Status: Pending.

LOW **L-06**

RESOLVED

# Redundant Queued State Execution Gate

CATEGORY Validation  
LOCATION VetoGovernor.sol  
REVIEW Main Review

## DESCRIPTION

The `VetoGovernor.execute` function diverges from the Velodrome implementation by accepting both `Succeeded` **and** `Queued` proposal states:

```
`solidity
require(
status == ProposalState.Succeeded || status == ProposalState.Queued,
"Governor: proposal not successful"
);
`
```

Yet the `state` function used currently used is the un-modified OpenZeppelin logic that never returns `ProposalState.Queued`; it can only evaluate to `Pending`, `Active`, `Succeeded`, `Defeated`, `Executed`, `Canceled` or `Vetoed`.

```
`solidity
if (_quorumReached(proposalId) && _voteSucceeded(proposalId)) {
return ProposalState.Succeeded;
} else {
return ProposalState.Defeated;
}
`
```

Because `Queued` is unreachable, the extra branch in `execute`` can be considered dead code.

## RECOMMENDATION

Consider removing the `ProposalState.Queued` check from the `VetoGovernor.execute` function.

## RESOLUTION

Status: Resolved.

LOW **L-07**

RESOLVED

# withdrawManaged Skips Pending Reward Claims

CATEGORY Warning  
LOCATION VotingEscrow.sol  
REVIEW Main Review

## DESCRIPTION

`RewardsDistributor.claim(mTokenId)` is the only path that moves weekly rebases from the global distributor into a managed NFT's `LockedManagedReward` contract. If a child veNFT is withdrawn before the parent managed lock has executed `claim` for the most recently finished epoch, that child's share of those yet-to-be-forwarded tokens is silently lost.

Execution flow:

1. A rebase accrues; `RewardsDistributor.claimable(mTokenId)` is non-zero but no claim call is made.
  2. The owner invokes `Voter.withdrawManaged(childId)` which call to:
    - `LockedManagedReward.earned(childId)` returns `zero` because the rebase has not been forwarded.
    - The child's weight is burned and permanently removed from future accounting.
  3. Later, when `RewardsDistributor.claim(mTokenId)` is finally called, the rebase arrives in `LockedManagedReward`, but the withdrawn child no longer participates in the weight distribution, so its rightful portion is redistributed to the remaining depositors.
- Therefore, any depositor who exits a managed lock without first triggering a parent-level `RewardsDistributor.claim` forfeits its pending epoch rewards forever.

## RECOMMENDATION

Require `withdrawManaged` to proceed only when `RewardsDistributor.claimable(mTokenId) == 0`, i.e. the manager has already forwarded the latest rebase.

## RESOLUTION

Status: Resolved.

LOW **L-08**

RESOLVED

# Managed NFTs Can Vote More Than Once Per Epoch

CATEGORY	Warning
LOCATION	Voter.sol
REVIEW	Main Review

## — DESCRIPTION

The owner of the managed NFT could break the invariant that an NFT can only vote once per epoch. Assume Bob owns the following 2 managed NFTs:

- Managed veNFT (called mNFTa) with 1 locked NFT (called INFTa).
- Managed veNFT (called mNFTb) with 1 locked NFT (called INFTb).
- The balance of INFTa and INFTb is the same.

Bob voted on `poolX` with mNFTa and mNFTb on the first hour of the epoch. At the last two hours of the voting windows of the current epoch, Bob changed his mind and decided to vote on the `poolY`. Under normal circumstances, the `onlyNewEpoch` modifier will prevent mNFTa and mNFTb from triggering the `Voter.vote` function because these two veNFTs have already voted in the current epoch and their `lastVoted` is set to a timestamp within the current epoch.

However, it is possible for Bob to bypass this control. Bob could call `Voter.withdrawManaged` function to withdraw INFTa and INFTb from mNFTa and mNFTb respectively. Since the weight becomes zero, the `lastVoted` for both mNFTa and mNFTb will be cleared. As a result, they will be allowed to re-vote in the current epoch.

Bob will call `Voter.depositManaged` to deposit INFTb into mNFTa and INFTa into mNFTb respectively to increase the weight of the managed NFTs.

Bob then calls `Voter.vote` with mNFTa and mNFTb to vote on `poolY`. Since the `lastVoted` is empty (cleared earlier), the `onlyNewEpoch` modifier will not revert the transaction.

## — RECOMMENDATION

To prevent any managed NFT owner from abusing this issue, consider disallowing re-voting in the current epoch once the last NFT has been withdrawn from a managed NFT.

## — RESOLUTION

Status: Resolved.

LOW L-09

PENDING

# Bribe And Fee Token Emissions Can Be Gamed By Users

CATEGORY Warning  
LOCATION Voter.sol  
REVIEW Main Review

## — DESCRIPTION

A user may vote or reset their vote once per epoch. Votes persist across epochs and once a user has distributed their votes among their chosen pools, the `poke` function may be called by any user to update the target user's decayed veNFT token balance. However, the `poke` function is not hooked into any of the reward distribution contracts. As a result, a user is incentivized to vote as soon as they create their lock and avoid re-voting in subsequent epochs. The amount deposited via `Reward._deposit` does not decay linearly as how it is defined under veToken mechanics. Therefore, users could continue to earn trading fees and bribes even after their lock has expired. Simultaneously, users can `poke` other users to lower their voting weight and maximize their own earnings.

## — RECOMMENDATION

Re-designing the `FeesVotingReward` and `BribeVotingRewards` contracts may be worth while decaying user deposits automatically.

## — RESOLUTION

Status: Pending.

LOW **L-10**

RESOLVED

# Wrong DOMAIN\_TYPEHASH Definition In VotingEscrow Contract

CATEGORY	Best Practices
LOCATION	VotingEscrow.sol
REVIEW	Main Review

## DESCRIPTION

The `delegateBySig` function implemented in the `VotingEscrow` contract is intended to allow off-chain signature delegation using the EIP-712, but the contract's hard-coded `DOMAIN_TYPEHASH` does not match the fields actually encoded in the domain separator. The code defines:

```
``solidity
bytes32 public constant DOMAIN_TYPEHASH =
keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");
`
a type hash for three fields (name , chainId , verifyingContract), yet in delegateBySig it
does:
`solidity
bytes32 domainSeparator = keccak256(
abi.encode(
DOMAIN_TYPEHASH,
keccak256(bytes(name)),
keccak256(bytes(version)),
block.chainid,
address(this)
)
);
`
which encodes four parameters (name , version , chainId , verifyingContract). Because the type
hash and the actual ABI-encoded layout diverge, any signature produced off-chain with the correct domain
(including version) will compute a different digest than the contract expects, causing every call to revert with
"invalid signature." This breaks all signature-based delegation and prevents any off-chain wallet or service
from delegating veNFT votes via signature.
```

**L-10** Wrong DOMAIN\_TYPEHASH Definition In VotingEscrow Contract

CONTINUED

**RECOMMENDATION**

According to the [EIP 712](https://eips.ethereum.org/EIPS/eip-712), in the [definition of domainSeparator](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md#definition-of-domainseparator), `string version` represents the current major version of the signing domain. Signatures from different versions should never be compatible. Consider updating the `DOMAIN_TYPEHASH` to include the `version` field:

```
``solidity
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,string
version,uint256 chainId,address verifyingContract)");
``
```

**RESOLUTION**

Status: Resolved.

LOW **L-11**

RESOLVED

# Forwarder EIP-7702 Compatability

CATEGORY Warning  
LOCATION Forwarder.sol: 91  
REVIEW Main Review

## DESCRIPTION

The `Forwarder` contract refunds any excess ETH to the signer of the transaction using a `transfer()` call. It assumes that's safe because if the account signed the transaction, it should be an EOA. However, after EIP-7702 was implemented, EOAs can have code as well. This means EOAs with code must have receive functions which logic execution cost doesn't surpass 2300 gas to safely use the Forwarder.

This can also be used for griefing in the following way:

- an EOA without receive functions uses the forwarder
- they send some value, but use a non refunding feature so they don't expect refund
- a malicious user frontruns their transaction by sending 1 wei to the `Forwarder`, triggers the refund and the transaction reverts.

## RECOMMENDATION

Warn the users about this limitation. You can also consider using `call()` instead of `transfer()`

## RESOLUTION

Status: Resolved.

LOW **L-12**

RESOLVED

# Killed Gauge May Accrue Rewards

CATEGORY Rewards

LOCATION Voter.sol: 386-391

REVIEW Main Review

## DESCRIPTION

`Voter.killGauge()` allows the admin to kill a gauge. Once killed, the gauge should stop accruing rewards. Whenever `_updateFor(gauge)` is executed, the gauge's reward is sent to the minter instead. However, `reviveGauge()` doesn't call `_updateFor(gauge)` before reviving the gauge. This creates a period of time between the last update for the killed gauge and the revival timestamp where the gauge is eligible for rewards. It's even possible that it hasn't been updated for the whole duration from killing to reviving. In result, the killed gauge will receive rewards.

## RECOMMENDATION

Update the gauge before reviving.

```
``diff
function reviveGauge(address _gauge) external {
+ _updateFor(_gauge);
if (_msgSender() != emergencyCouncil) revert NotEmergencyCouncil();
if (isAlive[_gauge]) revert GaugeAlreadyRevived();
isAlive[_gauge] = true;
emit GaugeRevived(_gauge);
}
``
```

## RESOLUTION

Status: Resolved.

LOW **L-13**

RESOLVED

## maxVotingNum can be bypassed

CATEGORY	Validation
LOCATION	Voter.sol
REVIEW	Main Review

### — DESCRIPTION

The `maxVotingNum` in the `Voter` contract can be bypassed by calling `poke()`. For example:

- `maxVotingNum = 60`
- NFT 1 votes for 60 pools
- the governor sets `maxVotingNum = 20`
- The owner of NFT 1 can now just keep increasing the lock time or the amount of their lock, call `poke()` and vote again for all of the 60 pools.

### — RECOMMENDATION

Consider allowing anyone to call `reset()` if `poolVote[_tokenId].length > maxVotingNum`. This will force the ve holder to vote again with the new restriction.

### — RESOLUTION

Status: Resolved.

LOW **L-14**

RESOLVED

# Gauge Rewards Are Lost When `totalWeight == 0`

CATEGORY	Rewards
LOCATION	Minter.sol, Voter.sol
REVIEW	Main Review

## DESCRIPTION

`Voter.notifyRewardAmount()` accepts weekly rewards from the minter even if `totalWeight == 0`. This will update the global index and in result all of the rewards will remain stuck in the contract because once a pool is voted for, its index will be set to the global index which has already been updated.

## RECOMMENDATION

Consider sending the rewards from the Minter only if there are votes for at least one pool:

```
``diff
+ if (voter.totalWeight() > 0)
address(psp).safeApprove(address(voter), _emission);
voter.notifyRewardAmount(_emission);
+ }
``
```

## RESOLUTION

Status: Resolved.

LOW **L-15**

RESOLVED

# AirdropDistributor's safeTransferFrom Enables DOS

CATEGORY	DoS
LOCATION	AirdropDistributor.sol
REVIEW	Main Review

## — DESCRIPTION

The `AirdropDistributor.distributeTokens` loop uses `ve.safeTransferFrom(address(this), _wallet, _tokenId)` for every recipient in `_wallets`. If any `_wallet` is a contract that does not implement `onERC721Received`, that single `safeTransferFrom` call will revert and bubble up, aborting the entire distribution and leaving all remaining recipients unpaid.

## — RECOMMENDATION

Switch from `safeTransferFrom` to the standard `transferFrom` for airdrops, so that sending into non-compliant contracts does not revert. This change will allow the distributor to continue looping past any address that cannot accept ERC-721 callbacks, ensuring no single bad recipient can halt the entire batch.

## — RESOLUTION

Status: Resolved.

LOW **L-16**

RESOLVED

# Missing Require Check In EpochGovernor.propose Function

CATEGORY	Validation
LOCATION	EpochGovernor.sol
REVIEW	Main Review

## — DESCRIPTION

The `propose` function in `GovernorSimple` does not verify that the `values` array contains exactly one element set to zero. It only checks `targets.length == 1` and `calldatas.length == 1`, but omits any assertion on values. As a result, a caller can submit a proposal with `values = [1]` (or any non-zero value), generating a distinct `proposalId`. That malformed proposal will always revert during `execute`, because it attempts to send ETH to the non-payable `nudge` function, but it does not block or interfere with properly formed proposals that use `values = [0]`. The missing check simply allows junk proposals that consume gas and clutter the proposal namespace.

## — RECOMMENDATION

Validate in `propose` that `values.length == 1` and that `values[0]` equals zero. Reject any proposal specifying a non-zero ETH transfer, ensuring `execute` can always successfully invoke `nudge` and allow the emission rate to update as intended.

## — RESOLUTION

Status: Resolved.

LOW **L-17**

RESOLVED

## Wrong Assumption In `Vote.withdrawManaged()`

CATEGORY Informational

LOCATION Voting.sol

REVIEW Main Review

### — DESCRIPTION

`Vote.withdrawManaged()` assumes that if the last locked token in a managed lock has been withdrawn, the managed lock has 0 weight.

```
``plain text
// If the NORMAL veNFT was the last tokenId locked into _mTokenId, reset vote as there is
// no longer voting power available to the _mTokenId. Otherwise, updating voting power to
accurately
// reflect the withdrawn voting power.
uint256 _weight = IVotingEscrow(ve).balanceOfNFTAt(_mTokenId, block.timestamp);
if (_weight == 0) {
 _reset(_mTokenId);
// clear out lastVoted to allow re-voting in the current epoch
delete lastVoted[_mTokenId];
} else {
 _poke(_mTokenId, _weight);
}
`
```

This won't be true most of the times because `VotingEscrow.withdrawManaged` decreases the amount of the managed lock by `_weight+_reward`. Because of the way rewards are distributed, users that withdraw will almost always leave unclaimed rewards. In addition to that, there will be a precision loss happening. As a result, when the lock is withdrawn from the managed lock, the total deducted rewards won't be equal to the total deposited rewards and the managed lock's `lastVoted` won't be reset.

It's also worth noting that this dust amount will compound over time and result in a larger `totalSupply` being reported, making it harder to reach `quorum` and `proposalThreshold` in `VetoGovernor`.

### — RECOMMENDATION

**L-17** Wrong Assumption In `Vote.withdrawManaged()`

CONTINUED

If resetting the `lastVote` mapping is something that really needs to happen, you can track the number of locks deposited in the managed locks and reset it when they become 0.

**— RESOLUTION**

Status: Resolved.

LOW **L-18**

RESOLVED

# No Minimum Duration For Locks

CATEGORY MEV  
LOCATION VotingEscrow.sol  
REVIEW Main Review

## — DESCRIPTION

`VotingEscrow` doesn't have a minimum duration restriction for creating locks. This means users can sandwich `Minter.updatePeriod()` calls:

1. Create a short lock with big amount in the previous block
2. Receive rewards
3. Withdraw the lock

Since `Minter.updatePeriod()` is permissionless, step 2 and 3 can be batched together in the same transaction. As a result, the user gets shares of the reward emissions with little to no commitment towards the system.

## — RECOMMENDATION

Consider adding a minimum duration for locks of for example 1 hour.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team2/pull/5>

## — RESOLUTION

Status: Resolved.

LOW **L-19**

RESOLVED

# Unclaimed Rewards Become Inaccessible

CATEGORY	DoS
LOCATION	RewardsDistributor.sol: 140
REVIEW	Main Review

## — DESCRIPTION

The `getReward` function in the `FreeManagedReward` and `VotingReward` contracts allow users to claim rewards for a specific token, similar to how the `claim` function in the `RewardsDistributor` contract operates. However, a potential issue arises when a user withdraws, merges or splits their token before claiming their accumulated rewards. In such cases, the rewards associated with the token become inaccessible, effectively leaving them stuck in the contract with no way for the user to retrieve them.

## — RECOMMENDATION

Clearly document this to ensure users are aware that they must claim all pending rewards before withdrawing, merging, or splitting their token. Consider implementing safeguards or warnings in the UI to help prevent accidental loss of claimable rewards.

## — RESOLUTION

Status: Resolved.

LOW **L-20**

RESOLVED

# killGauge Forwards Rewards to Minter

CATEGORY Rewards  
LOCATION Voter.sol: 372  
REVIEW Main Review

## DESCRIPTION

The `killGauge` function allows the `emergencyCouncil` to disable a gauge. However, it does not call `updateFor` on the gauge before killing it, meaning any pending rewards accrued by the gauge remain unaccounted for in the `claimable` mapping. Additionally, the function sends any rewards already recorded in `claimable` directly to the `minter` address instead of the gauge.

As a result, rewards that should have been distributed to users of the gauge are instead redirected to the minter address.

## RECOMMENDATION

Consider calling `_distribute` on the gauge before modifying its state, to ensure pending rewards are correctly distributed to the gauge rather than redirected to the minter. If any rewards remain after the call, send the leftovers to the minter address.

## RESOLUTION

Status: Resolved.

LOW L-21

RESOLVED

# Missing Require Check In EpochGovernor.propose Function

CATEGORY	Validation
LOCATION	EpochGovernor.sol
REVIEW	Main Review

## — DESCRIPTION

The `propose` function in `GovernorSimple` does not verify that the `values` array contains exactly one element set to zero. It only checks `targets.length == 1` and `calldatas.length == 1`, but omits any assertion on values. As a result, a caller can submit a proposal with `values = [1]` (or any non-zero value), generating a distinct `proposalId`. That malformed proposal will always revert during `execute`, because it attempts to send ETH to the non-payable `nudge` function, but it does not block or interfere with properly formed proposals that use `values = [0]`. The missing check simply allows junk proposals that consume gas and clutter the proposal namespace.

## — RECOMMENDATION

Validate in `propose` that `values.length == 1` and that `values[0]` equals zero. Reject any proposal specifying a non-zero ETH transfer, ensuring `execute` can always successfully invoke `nudge` and allow the emission rate to update as intended.

## — RESOLUTION

Status: Resolved.

LOW L-22

PENDING

# Team Controls Proposal Threshold Without Governance

CATEGORY	Warning
LOCATION	ProtocolGovernor.sol
REVIEW	Main Review

## DESCRIPTION

In `ProtocolGovernor`, the `setProposalNumerator` function allows only the `team` address (queried from `IVotingEscrow(ve).team()`) to update the `proposalNumerator` value, which directly sets the `proposalThreshold` as a percentage of total supply (`proposalNumerator / PROPOSAL_DENOMINATOR`, starting at 0.02% and capped at 5%). This bypasses the governance process entirely, as proposals and votes from token holders cannot influence or vote on changes to this key parameter.

Relevant code snippet:

```
``solidity
function setProposalNumerator(uint256 numerator) external {
 if (msg.sender != IVotingEscrow(ve).team()) revert NotTeam();
 if (numerator > MAX_PROPOSAL_NUMERATOR) revert ProposalNumeratorTooHigh();
 proposalNumerator = numerator;
}
``
```

## RECOMMENDATION

Route `setProposalNumerator` through governance proposals, replacing the team check with `onlyGovernance` modifier to ensure changes require voter approval.

## RESOLUTION

Status: Pending.

LOW **L-23**

RESOLVED

# Incorrect prior Results In Reward Contract

CATEGORY	Unexpected Behavior
LOCATION	Reward.sol
REVIEW	Main Review

## DESCRIPTION

Reward.sol records balance and supply checkpoints for each epoch. It also exposes two view functions - `getPriorBalanceIndex()` and `getPriorSupplyIndex()` - which return the index of the last checkpoint before or at a given timestamp.

This result will be inaccurate because a new checkpoint is created for a each epoch, but not each timestamp. For example:

- An epoch is 7 days long
- Alice deposits 300 on day 3. A checkpoint for epoch 1 is created with ( `balance = 300; timestamp = day 3` ).
- Alice deposits 500 on day 8. A checkpoint for epoch 2 is created with ( `balance = 800; timestamp = day 8` ).
- Alice deposits 1000 more on day 10. The same checkpoint for epoch 2 is updated with ( `balance = 1800; timestamp = day 10` )
- Now if we call `getPriorBalanceIndex()` for `timestamp = day 9`, the function will return index 0, where the balance of Alice is `300`, even though it was `800` in reality.

## RECOMMENDATION

Currently, these functions are called with `_currTs + DURATION - 1`, where `_currTs` is a week cursor value. This is safe because we are looking at the last checkpoint in the given epoch. Any further checkpoints will create a new record since we would be in a different epoch.

However, you should be really careful with the usage of these functions in the future.

## RESOLUTION

Status: Resolved.

LOW **L-24**

ACKNOWLEDGED

## Potential OOG Revert For withdrawManaged

CATEGORY Gas Optimization

LOCATION Reward.sol

REVIEW Main Review

### — DESCRIPTION

When users call `withdrawManaged()`, which will invoke `lockedRewardManaged.earned()` twice and will loop through all the unclaimed epochs for the user to calculate the earned amount twice. This creates a dangerous possibility of the transaction going over the block gas limit and as result stuck lock for the user.

### — RECOMMENDATION

Consider informing users that they should regularly redeposit their locks.

### — RESOLUTION

Status: Acknowledged.

LOW **L-25**

RESOLVED

# Some Proposals Cannot Be Executed

CATEGORY Unexpected Behavior  
LOCATION GovernanceSimple.sol  
REVIEW Main Review

## DESCRIPTION

Any proposals created in the interval `[weekCursor - votingDelay - 1; weekCursor)`, where `weekCursor` is the timestamp at which a new epoch begins, cannot be executed. Currently, that value is `901`. The reason is that for a proposal to be executed, at least `votingDelay + votingPeriod` have to pass. Since `votingPeriod` is 1 week, such proposals will become executable at least `1 week + 15 minutes + 1` seconds later. For example:

- We create a proposal 901 seconds before week 0 ends. Its `proposalId` is calculated with `epochStart = 1`.
- After 900 seconds pass, users can vote for it
- After `1 week + 1` more seconds pass, the proposal becomes executable. However, we are now at the beginning of week 2
- `GovernanceSimple.execute()` will use week 2 when calculating the `proposalId`. As a result, our old proposal becomes inaccessible

## RECOMMENDATION

You can either:

- document this behavior so users know these proposals always fail
- revert the call to `propose()` if we are in the failing interval

## RESOLUTION

Status: Resolved.

LOW L-26

RESOLVED

# Flash-Vote Bypass via split() and merge()

CATEGORY	Validation
LOCATION	VotingEscrow.sol
REVIEW	Main Review

## DESCRIPTION

The `VotingEscrow.split` function mints two brand-new veNFTs in the same block, each inheriting voting power from the original. Because these new token IDs have never been transferred, their `ownershipChange` remains zero, meaning the flash-transfer protection (which zeroes out voting power for any NFT that changed hands in the current block) does not apply. Downstream voting logic ( `Voter.vote` → `balanceOfNFTAt` ) uses the raw checkpoint function, so a user can:

1. Acquire veNFT #1.
2. In the same block, call `split(1, fullAmount)`.
3. Immediately call `Voter.vote` with one of the fresh IDs (#2 or #3), capturing full voting power.

This protection can also be bypassed by minting a new veNFT and immediately merging it into an existing one. Since the `merge` function does not update `ownershipChange` for the destination `tokenId`, the merged veNFT can vote in the same block, defeating the purpose of the flash protection.

## RECOMMENDATION

Consider updating the `split` and `merge` functions to set `ownershipChange[newId] = block.number` for each token created.

## RESOLUTION

Status: Resolved.

LOW L-27

RESOLVED

# Unnecessary Check In GovernanceSimple

CATEGORY Gas Optimization  
LOCATION GovernorSimple.sol: 279  
REVIEW Main Review

## DESCRIPTION

`GovernanceSimple.propose()` has two `require` statements about the `targets` array - the first one requires the length to be exactly 1 and the second one requires it to not be 0. The second one is unnecessary, since if the length was 0, the first one would have caught it.

## RECOMMENDATION

Remove the second `require()` statement

```
```diff
```

```
require(targets.length == 1, "GovernorSimple: only one target allowed");  
require(address(targets[0]) == minter, "GovernorSimple: only minter allowed");  
require(calldatas.length == 1, "GovernorSimple: only one calldata allowed");  
require(bytes4(calldatas[0]) == IMinter.nudge.selector, "GovernorSimple: only nudge allowed");
```

```
bytes32 epochStart = bytes32(ProtocolTimeLibrary.epochStart(block.timestamp) + (1 weeks));  
uint256 proposalId = hashProposal(targets, values, calldatas, epochStart);
```

```
- require(targets.length > 0, "GovernorSimple: empty proposal");  
```
```

## RESOLUTION

Status: Resolved.

LOW **L-28**

RESOLVED

# No Flash-Loan Protection In `balanceOfNFTAt`

CATEGORY Logical Error  
LOCATION `VotingEscrow.sol: 1070`  
REVIEW Main Review

## — DESCRIPTION

In the `VotingEscrow` contract, the `balanceOfNFT` function enforces a flash-loan protection preventing newly transferred veNFTs from being used for voting or gauge boosting in the same block by returning zero if `ownershipChange[_tokenId] == block.number`. However, the `balanceOfAtNFT` function that computes the NFT's voting power does not enforce the same ownership-change check. As a result, protocols or external calls that rely on them can see a non-zero voting balance even if the veNFT was transferred in the current block.

## — RECOMMENDATION

Align the same-block flash check in the `balanceOfAtNFT` function.

## — RESOLUTION

Status: Resolved.

LOW L-29

PENDING

# Growth Formula Rewards Early Lockers Disproportionately

CATEGORY Math  
LOCATION Minter.sol  
REVIEW Main Review

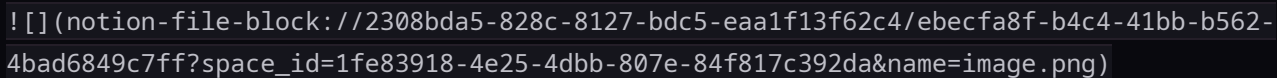
## DESCRIPTION

The current `Minter.calculateGrowth` implementation completely reverses Velodrome's reward logic, giving an outsized bonus to the very first lockers:

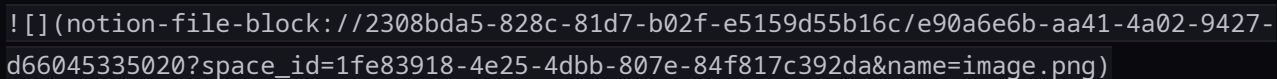
```
```solidity
// Original Velodrome: growth scales with locked share (Ve )
return ((((_minted * _veTotal) / _veloTotal) * _veTotal) / _veloTotal) * _veTotal / _veloTotal / 2;

// Current code: growth scales with circulating share ((Vtot-Ve) )
return (((_minted * (_aeroTotal - _veTotal)) / _aeroTotal) * (_aeroTotal - _veTotal)) / _aeroTotal / 2;
```
```

Velodrome:



Current formula:



- Launch behaviour: With `ve.totalSupplyAt(activePeriod-1) == 0``, week 0 growth equals 50% of emissions ( $\approx 5$  MPSPS). All of it accrues to the wallets that receive the permanent-lock NFTs immediately after, costing them nothing.

- Long-term behaviour:

- It does not incentivize locking because growth shrinks as the locked share rises, there is little incentive for the final 20–30% of holders to join the protocol once a majority has already locked.

- At  $\sim 57\%$  locked, Velodrome's cubic rule would pay more growth than this quadratic-circulating rule and therefore the protection for late lockers is weaker.

- Economic impact:

- Early inflation: epoch 0 mints  $\approx 15.8$  M tokens instead of 10 M (+50%).

- Governance concentration: early lockers rapidly accumulate supply, keeping governance

**L-29** Growth Formula Rewards Early Lockers Disproportionately

CONTINUED

control.

- APR whiplash: yields start extremely high, then decay sharply as more tokens lock.

**— RECOMMENDATION**

If the goal is Velodrome-style “lock-more-earn-more” incentives: revert to the cubic-locked formula. If fast boot-strapping is intentional: keep the new formula but:

- impose an upper cap on growth (e.g.,  $\leq X\%$  of current ve supply) to prevent extreme early windfalls.
- publicly document and justify the alternate incentive curve, and update economic models, emissions schedules and front-end displays to match the revised supply dynamics.

**— RESOLUTION**

Status: Pending.

LOW L-30

ACKNOWLEDGED

# Low-Decimal Tokens Can Be Griefed

CATEGORY Rewards  
LOCATION MultiRewards.sol: 250  
REVIEW Main Review

## DESCRIPTION

The `updateReward` function in the `MultiRewards` contract updates both `rewardPerTokenStored` and `lastUpdateTime`, which are used to calculate rewards for stakers. The `rewardPerTokenStored` value is computed using the formula:  $(\text{timeElapsed} * \text{rewardRate} * 1e18) / \text{totalSupply}$ . A precision loss edge case arises when  $(\text{timeElapsed} * \text{rewardRate} * 1e18)$  is less than `totalSupply`. In such cases, the division rounds down to zero, leaving `rewardPerTokenStored` unchanged. However, `lastUpdateTime` is still updated, meaning any reward for that duration is effectively lost. This results in zero yield for that period, despite rewards having accrued. Because the `updateReward` function can be triggered permissionlessly—for example, via the `getRewards` function—an attacker can repeatedly invoke it to continually suppress rewards for a given `_rewardsToken`.

## RECOMMENDATION

Consider modifying the reward update logic to use a higher scale for low-decimal tokens to minimize precision loss and reduce the potential for griefing attacks.

## RESOLUTION

Status: Acknowledged.

LOW **L-31**

RESOLVED

# Stale Proposals Can Be Revived Indefinitely

CATEGORY Logical Error  
LOCATION VetoGovernor.sol: 318  
REVIEW Main Review

## — DESCRIPTION

The `ProtocolGovernor` contract allows users to propose and vote on changes to core protocol parameters. If a proposal fails—either due to insufficient quorum or more “against” than “for” votes—it remains indefinitely active, with no permissionless way to finalize or expire it.

This means failed proposals can be revived at any time simply by adding more votes, regardless of how much time has passed or how governance sentiment may have shifted. While a `veto` function exists to manually finalize proposals, it relies on an active vetoer, which can be renounced—leaving no mechanism to clear out stale proposals.

## — RECOMMENDATION

Consider adding an expiry or permissionless finalization or expiry mechanism for failed proposals to prevent indefinite revival.

## — RESOLUTION

Status: Resolved.

LOW L-32

PENDING

# Bribe Rewards Distributed In Epochs With No Depositors Are Lost

CATEGORY Logical Error  
LOCATION BribeVotingReward.sol  
REVIEW Main Review

## — DESCRIPTION

Rewards that are sent to `BribeVotingReward` vaults are lost if there are no vault depositors for that epoch. Anyone can send bribe voting rewards to vaults by calling `BribeVotingReward.notifyRewardAmount`. This updates the mapping `tokenRewardsPerEpoch[token][epochStart]` which stores the token rewards per epoch with the newly added amount. Since the rewards are stored and distributed in their entirety by epochs, they will be lost and irrecoverable if there are no vault depositors for those epochs.

## — RECOMMENDATION

Consider implementing a new function `renotifyRewardAmount` allowing for undistributed rewards to be redistributed.

## — RESOLUTION

Status: Pending.

LOW **L-33**

ACKNOWLEDGED

# Precision Loss Causes Irrecoverable Rewards

|          |                   |
|----------|-------------------|
| CATEGORY | Rewards           |
| LOCATION | Rewards contracts |
| REVIEW   | Main Review       |

## — DESCRIPTION

The `_notifyRewardAmount` function in the `Gauge` contract processes incoming reward amounts by updating the `rewardRate` for the given reward token.

However, the calculation `rewardRate = amount / timeUntilNext` introduces precision loss, leaving a residual amount equal to `amount % timeUntilNext`. While the entire `reward` amount is transferred to the contract, a leftover remainder becomes permanently stuck. Over time, these unallocated "dust" amounts accumulate in the contract with no way to recover or redistribute them.

This issue can also affect other contracts with similar reward logic, such as `BribeVotingReward` and `FreeManagedReward`.

## — RECOMMENDATION

Implement a recover function, callable by an admin, to allow manual recovery of accumulated dust.

## — RESOLUTION

Status: Acknowledged.

LOW **L-34**

RESOLVED

# Valid Proposals Do Not Have Deadlines

CATEGORY Unexpected Behavior

LOCATION VetoGovernor.sol

REVIEW Main Review

## DESCRIPTION

Once a proposal passes in the `VetoGovernor`, it becomes eligible for execution. If the transaction fails, it can be retried repeatedly until it eventually succeeds. This behavior can lead to unexpected situations—for example, a function call that initially reverted might be retried and executed much later, even after governance has chosen a different course of action. The presence of the `relay()` function increases the likelihood of such scenarios. While proposals can currently be vetoed to prevent this, once `renounceVetoer()` is called, there will be no mechanism left to block their execution.

## RECOMMENDATION

Consider adding a period of time after which the successful proposal expires.

## RESOLUTION

Status: Resolved.

LOW **L-35**

RESOLVED

# Incorrect Logic In Voter/Distributor Check

CATEGORY Logical Error  
LOCATION ve.ts: 110-116  
REVIEW Main Review

## DESCRIPTION

The deployment script contains a logical error in the condition checking whether the voter and distributor addresses need to be updated in the `VotingEscrow` contract.

```
``typescript
if (
 (await votingEscrowContract.voter()) !== voter.address &&
 (await votingEscrowContract.distributor()) !== rewardsDistributor.address
) {
 await (await votingEscrowContract.setVoterAndDistributor(voter.address,
 rewardsDistributor.address)).wait();
} else {
 console.log("Voter and RewardsDistributor already set");
}
`
```

The condition uses `&&` (AND) instead of `\|\|` (OR), which means the `setVoterAndDistributor` function will only be called if both addresses are different from the current values. This is problematic because:

- If only one address needs updating, the function won't be called. The contract may end up with mixed old/new addresses
- With `skipIfAlreadyDeployed: true``, subsequent deployments may skip necessary updates

## RECOMMENDATION

Change the condition to use `\|\|` (OR) to ensure the function is called if either address needs updating:

## RESOLUTION

Status: Resolved.

LOW **L-36**

RESOLVED

# Expired Token Can Be Merged to Extend Lock

CATEGORY Logical Error  
LOCATION VotingEscrow.sol: 912  
REVIEW Main Review

## DESCRIPTION

The `merge` function allows two token positions to be combined into one by summing their amounts, using the later of the two lock end times, and burning one of the token IDs. However, the function does not check whether the from token has already expired.

This creates a bypass to the `increaseUnlockTime` function, which explicitly prevents extending the lock on an expired token by requiring that the current lock end is greater than `block.timestamp`. By merging an expired token into an active one, the expired token's lock can effectively be extended.

## RECOMMENDATION

Add a check in the `merge` function to ensure that the `from` token is not expired.

## RESOLUTION

Status: Resolved.

LOW **L-37**

RESOLVED

## EIP compliance issues

|          |                                                        |
|----------|--------------------------------------------------------|
| CATEGORY | Unexpected Behavior                                    |
| LOCATION | VotingEscrow.sol, GovernorSimple.sol, VetoGovernor.sol |
| REVIEW   | Main Review                                            |

### — DESCRIPTION

EIP721 compliance:

The `VotingEscrow.supportsInterface()` function returns `true` when it's called with the interface id for `ERC721` (`0x80ac58cd`). However, it doesn't implement it correctly because:

- `balanceOf(address)` doesn't revert when called with `address(0)`.
- `ownerOf()` doesn't revert for invalid ids.
- `transferFrom()` and `safeTransferFrom()` don't revert when NFTs are transferred to the 0 address.
- `getApproved(uint256)` doesn't revert when it receives invalid token id.

Users should not be able to transfer tokens to `address(0)` because that's how `VotingEscrow` determines if a token exists or not. Potential issues from that are:

- `tokenURI()` will revert because the new owner of the token is `address(0)`.
- `delegate()` will also revert if an external integrator tries delegating to an NFT that has been transferred to `address(0)`.

EIP165 compliance:

- Both `GovernorSimple` and `VetoGovernor` are able to receive `ERC721` and `ERC1155` tokens. However, their `supportsInterface()` function includes only `ERC1155` and not `ERC721`.

When external integrators check if the Governor contract is eligible to receive NFTs, they will get `false`. This result can lead to many unexpected outcomes, like the Governor not receiving NFTs that it should, reverting transactions and others depending on how the given system and the contract interact with each other.

- The governance contracts also implement the EIP6372 clock, but `supportsInterface` doesn't include that either.

### — RECOMMENDATION

Fix the issues mentioned in the EIP721 compliance section. For the EIP165 section, add the `IERC721Receiver.interfaceId` and `IERC6372.interfaceId` towards the supported ids.

---

**L-37** EIP compliance issues

CONTINUED

---

— **RESOLUTION**

Status: Resolved.

LOW **L-38**

PENDING

# Poke Can Be Blocked By Transfer

CATEGORY DoS  
LOCATION Voter.sol: 201  
REVIEW Main Review

## — DESCRIPTION

The `poke` function in the `Voter` contract updates a user's voting power by resetting their old votes and applying new ones based on their current locked balance. It fetches this updated weight using `balanceOfNFT` from the `VotingEscrow` contract.

The `balanceOfNFT` function includes a flash-loan protection mechanism that returns zero if the token's ownership changed in the current block (`ownershipChange[_tokenId] == block.number`). While this check serves its intended purpose, it can be exploited: a user can front-run a `poke` call by transferring their token in the same block, causing `balanceOfNFT` to return zero. As a result, the internal `_vote` call in `poke` reverts due to the zero-weight.

This effectively allows users to block the `poke` function, preserving outdated vote weights and continuing to receive rewards and influence beyond what their actual balance should allow.

## — RECOMMENDATION

Consider implementing safeguards to prevent transfers from interfering with `poke`, or update the `poke` logic to better handle zero-weight cases triggered by same-block transfers.

## — RESOLUTION

Status: Pending.

LOW L-39

PENDING

# Team Controls Proposal Threshold Without Governance

|          |                      |
|----------|----------------------|
| CATEGORY | Warning              |
| LOCATION | ProtocolGovernor.sol |
| REVIEW   | Main Review          |

## DESCRIPTION

In `ProtocolGovernor`, the `setProposalNumerator` function allows only the `team` address (queried from `IVotingEscrow(ve).team()`) to update the `proposalNumerator` value, which directly sets the `proposalThreshold` as a percentage of total supply (`proposalNumerator / PROPOSAL_DENOMINATOR`, starting at 0.02% and capped at 5%). This bypasses the governance process entirely, as proposals and votes from token holders cannot influence or vote on changes to this key parameter.

Relevant code snippet:

```
``solidity
function setProposalNumerator(uint256 numerator) external {
 if (msg.sender != IVotingEscrow(ve).team()) revert NotTeam();
 if (numerator > MAX_PROPOSAL_NUMERATOR) revert ProposalNumeratorTooHigh();
 proposalNumerator = numerator;
}
`
```

Same applies also to the `setTeamRate` and `toggleSplit` functions.

## RECOMMENDATION

Route `setProposalNumerator`, `setTeamRate` and `toggleSplit` functions through governance proposals, replacing the team check with `onlyGovernance` modifier to ensure changes require voter approval.

## RESOLUTION

Status: Pending.

LOW **L-40**

RESOLVED

# withdrawManaged Skips Pending Reward Claims

CATEGORY Warning  
LOCATION VotingEscrow.sol  
REVIEW Main Review

## DESCRIPTION

`RewardsDistributor.claim(mTokenId)` is the only path that moves weekly rebases from the global distributor into a managed NFT's `LockedManagedReward` contract. If a child veNFT is withdrawn before the parent managed lock has executed `claim` for the most recently finished epoch, that child's share of those yet-to-be-forwarded tokens is silently lost.

Execution flow:

1. A rebase accrues; `RewardsDistributor.claimable(mTokenId)` is non-zero but no claim call is made.
2. The owner invokes `Voter.withdrawManaged(childId)` which call to:
  - `LockedManagedReward.earned(childId)` returns `zero` because the rebase has not been forwarded.
  - The child's weight is burned and permanently removed from future accounting.
3. Later, when `RewardsDistributor.claim(mTokenId)` is finally called, the rebase arrives in `LockedManagedReward`, but the withdrawn child no longer participates in the weight distribution, so its rightful portion is redistributed to the remaining depositors.

Therefore, any depositor who exits a managed lock without first triggering a parent-level `RewardsDistributor.claim` forfeits its pending epoch rewards forever.

## RECOMMENDATION

Require `withdrawManaged` to proceed only when `RewardsDistributor.claimable(mTokenId) == 0`, i.e. the manager has already forwarded the latest rebase.

## RESOLUTION

Status: Resolved.

LOW **L-41**

PENDING

# Zero Locked Supply Allows Bypass Of Quorum

CATEGORY Warning  
LOCATION config.json  
REVIEW Main Review

## DESCRIPTION

Based on the provided deployment `config.json`, the Minter contract is initialized with zero locked tokens. This creates a issue in the ProtocolGovernor's quorum calculation:

```
``plain text
function quorum(uint256 timepoint) public view virtual override returns (uint256) {
return (token.getPastTotalSupply(timepoint) * quorumNumerator(timepoint)) /
quorumDenominator();
}
`
```

Since `getPastTotalSupply` returns zero, the calculated quorum is also zero. This means any proposal can pass with zero votes, so long as it is not actively voted against during the voting period.

An attacker could propose to set the quorum numerator to zero, or use `relay`` to execute arbitrary transactions.

## RECOMMENDATION

During deployment, ensure hat a non-zero amount of tokens are locked. This guarantees a meaningful quorum threshold is enforced from the start, preventing governance from being bypassed due to a zero total supply.

## RESOLUTION

Status: Pending.

LOW **L-42**

RESOLVED

# Whitelisted Rewards Persist After Removal

|          |                           |
|----------|---------------------------|
| CATEGORY | Logical Error             |
| LOCATION | BribeVotingReward.sol: 12 |
| REVIEW   | Main Review               |

## DESCRIPTION

The `BribeVotingReward` and `FreeManagedReward` contracts restrict the `notifyRewardAmount` function to accept only tokens that are either already in the `isReward` mapping or currently whitelisted by the `Voter` contract. If a new token is whitelisted in the `Voter` contract, it can be added to the `isReward` mapping upon first use.

However, once a token has been added to `isReward`, the `notifyRewardAmount` function no longer checks whether the token is still whitelisted in the `Voter` contract. This means that if a token is later removed from the whitelist—due to governance decisions—it can still be used to add rewards. Additionally, there is no function to modify tokens in the `isReward` mapping, making the status effectively permanent.

As a result, tokens explicitly removed by governance can continue being distributed as rewards, bypassing intended access restrictions. If a token was removed for being malicious or unstable, this loophole poses a broader security and trust risk. Over time, the `rewards` array may also become bloated with stale or invalid tokens, further reducing transparency and control over what tokens are being used for rewards.

## RECOMMENDATION

Consider updating the `notifyRewardAmount` logic to always check the token's current whitelist status or implementing a way to modify tokens in the `isReward` mapping and `rewards` array when they are no longer whitelisted.

## RESOLUTION

Status: Resolved.

## INFORMATIONAL I-01

PENDING

# Unit-mismatch In `_claimFees` Can Prevent Fee Distribution

|          |                   |
|----------|-------------------|
| CATEGORY | Logical Error     |
| LOCATION | Gauge.sol: 91, 98 |
| REVIEW   | Main Review       |

## DESCRIPTION

The `Gauge` contract releases swap fees to the `feesVotingReward` bribe only when the local variables `fees0` or `fees1` exceed the constant `DURATION`, which is fixed at 7 days expressed in seconds ( `604800` ).

```
```solidity
```

```
// Gauge.sol (_claimFees)
```

```
uint256 internal constant DURATION = 7 days; // 604800
```

```
if (_fees0 > DURATION) { // token-amount > 604800 (seconds)
```

```
fees0 = 0;
```

```
_token0.safeApprove(feesVotingReward, _fees0);
```

```
IReward(feesVotingReward).notifyRewardAmount(_token0, _fees0);
```

```
} else {
```

```
fees0 = _fees0;
```

```
}
```

```
if (_fees1 > DURATION) {
```

```
fees1 = 0;
```

```
_token1.safeApprove(feesVotingReward, _fees1);
```

```
IReward(feesVotingReward).notifyRewardAmount(_token1, _fees1);
```

```
} else {
```

```
fees1 = _fees1;
```

```
}
```

```
``
```

`_fees0` and `_fees1` are ERC20 token amounts, not time values. Unless a pool accrues more than `604800` base-units of the fee token between two calls, the `if` condition will never be satisfied. In practice most tokens use 18 decimals so the threshold is 0.000604800 tokens, which is trivially passed, but for low-decimals or LPs with small volume the opposite occurs: fees accumulate without limit because the comparison is meaningless.`

I-01 Unit-mismatch In `_claimFees` Can Prevent Fee Distribution

CONTINUED

RECOMMENDATION

To correct the logic treat the threshold in the same unit as the variable being compared, or make the distribution time-based rather than amount-based. A minimal fix is to replace `DURATION` with a constant that represents a token-amount, for example `MIN_FEE_TO_DISTRIBUTE = 1e18`, or to distribute on a timer:

```
``solidity
uint256 public lastFeeCheckpoint;
uint256 public constant MIN_INTERVAL = 1 hours;

if (block.timestamp >= lastFeeCheckpoint + MIN_INTERVAL && _fees0 > 0) {
    fees0 = 0;
    _token0.safeApprove(feesVotingReward, _fees0);
    IReward(feesVotingReward).notifyRewardAmount(_token0, _fees0);
    lastFeeCheckpoint = block.timestamp;
}
``
```

Either approach removes the unit mismatch and ensures that collected LP fees reach their intended recipients.

RESOLUTION

Status: Pending.

INFORMATIONAL I-02

PENDING

Incorrect Time-Constant Comparison In `_distribute()`

CATEGORY Logical Error

LOCATION Voter.sol

REVIEW Main Review

DESCRIPTION

In the `Voter._distribute` function, the code requires that the accrued reward balance `_claimable` exceed both the remaining gauge capacity and the constant `DURATION`, which is defined as seven days (604800 seconds):

```
``solidity
if (_claimable > IGauge(_gauge).left() && _claimable > DURATION)
```

Because `DURATION` represents a time interval rather than a token quantity, any gauge with fewer than `604800`` tokens pending will never satisfy the second condition.

This misalignment of units appears to stem from a misplaced copy of a time-based check, but in the context of token distribution it prevents any legitimate payouts under the seven-day amount.

RECOMMENDATION

Replace the existing conditional so that distribution occurs whenever there is a positive claimable balance above the gauge's leftover capacity. For example:

```
``solidity
- if (_claimable > IGauge(_gauge).left() && _claimable > DURATION) {
+ if (_claimable > IGauge(_gauge).left()) {
claimable[_gauge] = 0;
address(rewardToken).safeApprove(_gauge, _claimable);
IGauge(_gauge).notifyRewardAmount(_claimable);
address(rewardToken).safeApprove(_gauge, 0);
emit DistributeReward(_msgSender(), _gauge, _claimable);
}
```

I-02 Incorrect Time-Constant Comparison In `_distribute()`CONTINUED

— RESOLUTION

Status: Pending.

INFORMATIONAL I-03

PENDING

Missing Contract-wallet Delegation Support

CATEGORY Warning
LOCATION VotingEscrow.sol
REVIEW Main Review

DESCRIPTION

The `delegateBySig` function in `VotingEscrow` allows delegation via an ECDSA signature recovered with `ecrecover`, but it rejects any attempts from smart-contract wallets because it never checks the ERC-1271 `isValidSignature` interface. As a result, a user whose veNFT is held in a Gnosis Safe, Argent, or any account that signs via a contract can not delegate by signature. This limitation forces contract-wallet holders into on-chain transactions for every delegation change, undermining meta-transaction UX and discouraging participation from institutional or multisig users.

RECOMMENDATION

Extend `delegateBySig` to first attempt an `isValidSignature(digest, signature)` call on the recovered address if `ecrecover` returns an address with no private-key. Only fall back to the existing on-chain approval check if the contract does not implement ERC-1271. This will allow multisig and other contract wallets to delegate seamlessly via signed messages.

RESOLUTION

Status: Pending.

INFORMATIONAL I-04

PENDING

Unused Imports

CATEGORY	Warning
LOCATION	VotingEscrow.sol
REVIEW	Main Review

DESCRIPTION

In the `VotingEscrow` contract a number of files are imported but never actually referenced in the code. For example, the top of the file includes

```
``solidity
import {Base64} from "@solady/utils/Base64.sol";
import {Ownable} from "@solady/auth/Ownable.sol";
import {IERC20} from "@openzeppelin/contracts/interfaces/IERC20.sol";
``
```

yet none of these types or libraries is ever invoked. Unused imports needlessly bloat the source and the compiler's dependency graph. They obscure which modules actually drive core logic and slightly increase contract metadata size, which flows into artifact and deployment overhead.

RECOMMENDATION

Prune all imports that are not referenced by the contract. In particular, remove `Base64`, `Ownable` and `IERC20` from the import list. Keeping only the truly used interfaces and libraries will make the codebase easier to navigate, minimize bytecode metadata and reduce the risk of carry-along vulnerabilities from unused modules.

RESOLUTION

Status: Pending.

INFORMATIONAL I-05

PENDING

Transient Storage Can Be Used For The ReentrancyGuard

CATEGORY Warning
LOCATION Global
REVIEW Main Review

— DESCRIPTION

The library [ReentrancyGuard] (<https://github.com/vectorized/solady/blob/main/src/Utils/ReentrancyGuard.sol>) is used. There is also a version that uses transient storage that is more efficient.

— RECOMMENDATION

Consider using [ReentrancyGuardTransient] (<https://github.com/vectorized/solady/blob/main/src/Utils/ReentrancyGuardTransient.sol>). Then check the chain on which the contract are deployed support transient storage.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-06

PENDING

veNFT Splitting Is Disabled By Default

CATEGORY	Configuration
LOCATION	VotingEscrow.sol
REVIEW	Main Review

DESCRIPTION

The `split` function in `VotingEscrow` lets a veNFT owner carve a single lock into two separate NFTs, but this capability is turned off for everyone until the on-chain team explicitly enables it. Under the hood, the contract checks

```
``solidity
if (!canSplit[owner] && !canSplit[address(0)]) revert SplitNotAllowed();
```

and by default both `canSplit[owner]` and the global `canSplit[address(0)]` are `false`. As a result, even if you hold a large lock and want to create smaller tranches for portfolio management, you cannot split your NFT until the team calls

```
`solidity
votingEscrow.toggleSplit(address(0), true);
``
```

RECOMMENDATION

Merely informative. Consider adding the `votingEscrow.toggleSplit(address(0), true);` call to your `ve.ts` script if this is an intended functionality you want to include since the protocol deployment.

RESOLUTION

Status: Pending.

INFORMATIONAL I-07

PENDING

Missalignment With Velodrome Tokenomics

CATEGORY Warning
 LOCATION RewardsDistributor.sol
 REVIEW Main Review

DESCRIPTION

A review of the `Minter` contract against the official Velodrome specification (see [SPECIFICATION.md#minter](https://github.com/velodrome-finance/contracts/blob/main/SPECIFICATION.md#minter)) reveals four key discrepancies that are purely informational but may confuse auditors or integrators:

Initial Weekly Emission

- Spec: 15 million PSPS per epoch

- Code:

```
``solidity
weekly = 10_000_000 * 1e18; // 10 million
```

Impact: integrations expecting 15 M on day one will see 10 M instead.

Growth vs. Decay Schedule

- Spec: emissions decay at 1% per epoch from the start

- Code:

```
`solidity
if (epochCount < 15) {
weekly = (weekly * WEEKLY_GROWTH) / MAX_BPS; // +3% for first 15 epochs
} else {
weekly = (weekly * WEEKLY_DECAY) / MAX_BPS; // -1% thereafter
}
```

Impact: emissions initially increase 3% weekly for 15 weeks, contrary to a constant 1% decline.

Tail-Emission Trigger

- Spec: tail mode begins once weekly falls below ~6 million

- Code:

```
`solidity
uint256 public constant TAIL_START = 8_969_150 * 1e18; // ~8.97 M
bool _tail = weekly < TAIL_START;
``
```

I-07 Missalignment With Velodrome Tokenomics

CONTINUED

Impact: tail schedule activates later than documented (~92 vs. ~134 epochs).

Initial Tail Rate

- Spec: 30 basis points (0.30%) initial tail rate

- Code:

```
` ` solidity
uint256 public tailEmissionRate = 67; // 67 bps = 0.67%
` `
```

Impact: tail emissions start at 0.67% instead of 0.30%.

— RECOMMENDATION

This issue is informational: update any external documentation or auditor notes to reflect the on-chain parameters as implemented, and refer to the official spec in [SPECIFICATION.md#minter] (<https://github.com/velodrome-finance/contracts/blob/main/SPECIFICATION.md#minter>) for the source of truth.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-08

PENDING

Unused _proposalProposer Internal View Function

CATEGORY	Warning
LOCATION	GovernorSimple.sol, VetoGovernor.sol
REVIEW	Main Review

— DESCRIPTION

In the `VetoGovernor` and `GovernorVotes` contracts, there is an internal helper

```
``solidity
```

```
function _proposalProposer(uint256 proposalId) internal view returns (address) {  
    return _proposals[proposalId].proposer;  
}
```

that simply reads the `proposer` field from the `Proposal` struct. This function is never exposed externally and its single-line body adds no abstraction or reuse benefit, since any consumer within the contract could call `_proposals[proposalId].proposer` directly and off-chain tools already learn the `proposer` from the `ProposalCreated` event.

— RECOMMENDATION

Consider removing the `_proposalProposer` function from the `VetoGovernor` and `GovernorVotes` contracts.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-09

PENDING

Best Practices For PSPS Reward Distribution For PurrLPStakingForPoints Stakers

CATEGORY	Warning
LOCATION	PurrLPStakingForPoints.sol
REVIEW	Main Review

DESCRIPTION

The current launch scripts reference an `airdrop.json` file intended to list wallet/amount pairs for a PSPS airdrop, but no logic exists that actually produces that file or decides how the total airdrop pool will be split among the four upcoming `PurrLPStakingForPoints` contracts (USDT0/USDE, USDT0/feUSD, WHYPE/USDT0, WHYPE/UETH).

Because each LP token will have a different value, an equal 25%÷25%÷25%÷25% split would over-reward holders of the least valuable LP token and under-reward those who provided the highest valuable LP tokens.

RECOMMENDATION

Derive every pool's share of the airdrop from its **relative dollar value locked (TVL)** at a well-defined snapshot block. If pool A holds \$10000 in LP Tokens staked and pool B holds \$20000, pool B should receive twice the PSPS that pool A receives. Compute each LP token's price safely.

RESOLUTION

Status: Pending.

INFORMATIONAL I-10

PENDING

Accounts Can Own Multiple Managed NFTs

CATEGORY Informational
LOCATION IVotingEscrow.sol: 214
REVIEW Main Review

DESCRIPTION

`VotingEscrow.createManagedLockFor()` will create a new managed NFT and assign it to a given address. The function can be used multiple times to mint many managed NFTs to that same address. This is opposite to what's stated in the NatSpec of the function which says: `Throws if address already owns a managed NFT.`

RECOMMENDATION

If the intention is to allow addresses to own multiple managed NFTs, remove that part of the NatSpec.

RESOLUTION

Status: Pending.

INFORMATIONAL I-11

PENDING

Deceiving Comment In split Function

CATEGORY Informational
LOCATION IVotingEscrow.sol: 406
REVIEW Main Review

DESCRIPTION

One of the comments in the NatSpec for the `VotingEscrow.split()` function says if the party calling the function is approved for that token, they won't be able to operate on the newly created tokens.

If this is called by approved, approved will not have permissions to manipulate the newly created veNFTs

While true for normal approvals, this statement is wrong for operators calling the `split` function - they will still be able to operate with the newly created NFTs because the same owner will receive them.

RECOMMENDATION

Consider modifying the comment to make it clear that approved operators will be able to modify the newly created NFTs.

RESOLUTION

Status: Pending.

INFORMATIONAL I-12

PENDING

Transfer to Zero Address Allowed

CATEGORY	Logical Error
LOCATION	VotingEscrow.sol: 364
REVIEW	Main Review

DESCRIPTION

Transfers of veNFT to the zero address are not prevented. Unlike a proper `_burn`, when transferring a veNFT to the zero address, no checkpoint is created, so bias and slope remain in global checkpoint history until lock expiry, inflating total voting power.

Rewards accruing to the veNFT (in the zero address) can still be claimed, as the fee distributor does not check for zero address ownership. However, once claimed, the underlying assets become permanently stuck in the VotingEscrow contract.

Additionally, PSPS `supply` in VotingEscrow is not reduced, leading to incorrect accounting in external contracts relying on that variable.

RECOMMENDATION

In `_transferFrom`, add an explicit check to reject `_to == address(0)`.

RESOLUTION

Status: Pending.

INFORMATIONAL I-13

PENDING

Request type is not closed properly

CATEGORY Best Practices
LOCATION Forwarder.sol: 113
REVIEW Main Review

DESCRIPTION

`Forwarder.registerRequestType()` allows users registering their custom request type with additional `suffixData`. The `requestType` string is opened with a `(`, but it's never closed. This means it's on the user to provide a `suffixData` ending with `)`, otherwise the EIP-712 signing procedure won't work correctly.

RECOMMENDATION

Consider closing the `requestType` string properly, so the user can only provide the data they need.

RESOLUTION

Status: Pending.

INFORMATIONAL I-14

PENDING

Gauges With `isPool = false` Never Receive Fees

CATEGORY Gas Optimization
LOCATION Gauge.sol, Voter.sol
REVIEW Main Review

DESCRIPTION

`Gauge._claimFees()` won't send any rewards to `FeeVotingRewards` if `isPool == false`. While this is intended, it's worth noting it creates some side effects, like paying more gas on `createGauge` because an unused contract is deployed, spending more gas on voting because `deposit/withdraw` are called on that contract even though it won't generate rewards.

RECOMMENDATION

Acknowledge the gas cost increases mentioned in the report.

RESOLUTION

Status: Pending.

INFORMATIONAL I-15

PENDING

transfer And unlock Constraints

CATEGORY	Informational
LOCATION	VotingEscrow.sol
REVIEW	Main Review

— DESCRIPTION

When a permanent lock is unlocked, its delegation is deleted. A delegation can only be deleted if the delegator NFT wasn't transferred in the same block. As a result, if a user transfers their NFT, they cannot unlock it within that same block. This behavior is important to note for external integrators.

— RECOMMENDATION

Document that behavior.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-16

PENDING

lockPermanent can be applied to voting tokens

CATEGORY Informational
LOCATION VotingEscrow.sol
REVIEW Main Review

DESCRIPTION

`VotingEscrow.lockPermanent()` doesn't check if the token has voted, which means that actively voting tokens can be changed to permanent locks.

RECOMMENDATION

Acknowledge this behavior.

RESOLUTION

Status: Pending.

INFORMATIONAL I-17

PENDING

Auto-compound Set To True Disables Gauge Fee Stream

CATEGORY Warning
LOCATION Bribes.sol
REVIEW Main Review

DESCRIPTION

When `_AUTO_COMPOUND_FEES_` is set to `true` in a `PurrLP` clone, the pool's fee-accrual mechanism places all swap fees directly back into the pool reserves, bypassing the per-address accounting indexes that drive

```
claimFees:
`solidity
function _updateBaseFees(uint256 amount) internal returns (uint256) {
if (!implementation._AUTO_COMPOUND_FEES_()) { // ← never executed
_BASE_TOKEN_.safeTransfer(fees, amount);
index0 += (amount * 1e18) / totalSupply();
} else if (_treasuryFeeBips > 0) {
_BASE_TOKEN_.safeTransfer(_treasury, treasuryAmount);
}
}
`
```

Because `index0` and `index1` remain constant, `claimFees` always returns `(0, 0)`:

```
`solidity
uint256 _delta0 = index0 - supplyIndex0[recipient]; // always zero
uint256 _delta1 = index1 - supplyIndex1[recipient]; // always zero
`
```

The `Gauge` contract relies exclusively on

```
`solidity
(claimed0, claimed1) = IPurrLP(stakingToken).claimFees();
`
```

to harvest these fees. With every call returning zero, the `Gauge`'s internal fee buffers never fill, no `IReward(feesVotingReward).notifyRewardAmount` is triggered and the entire downstream fee-sharing stream to the `FeesVotingReward` contract is silently cut off forever. Emissions still flow, but the protocol component that voters, dashboards and external participants expect, "swap fees distributed every epoch", simply vanishes. The value of those fees is permanently locked into the LP's reserves of the `Gauge`` contract instead of

I-17 Auto-compound Set To True Disables Gauge Fee Stream

CONTINUED

reaching tokenholders who stake or vote, resulting in a non-recoverable revenue loss and broken economic assumptions for the protocol. Switching `_AUTO_COMPOUND_FEES_` back to `false` cannot recover fees that were already compounded.

— RECOMMENDATION

Merely informative. Consider never setting `_AUTO_COMPOUND_FEES_` to true in an underlying `PurrLp` pool supported by a `Gauge`.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-18

PENDING

External Functions Naming

CATEGORY Best Practices

LOCATION Reward.sol

REVIEW Main Review

DESCRIPTION

The external `_deposit()` and `_withdraw()` functions in the `Reward.sol` contract are unconventionally prefixed with underscore, which confused the reader that they are internal functions.

RECOMMENDATION

Consider removing the underscore of their names.

RESOLUTION

Status: Pending.

INFORMATIONAL I-19

PENDING

increaseAmount () Donates To Rewards

CATEGORY	Informational
LOCATION	VotingEscrow.sol
REVIEW	Main Review

— DESCRIPTION

`VotingEscrow.increaseAmount()` allows an approved party for the managed lock to increase the locked amount of PSPS. However, the amount sent is distributed as rewards to the depositors for the given lock, which means the user will potentially lose part of these tokens or all of them if they haven't deposited into the lock themselves.

— RECOMMENDATION

Make sure the users are aware of this behavior.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-20

PENDING

Immediate Execution Lacks Timelock Delay

CATEGORY Warning
LOCATION VetoGovernor.sol
REVIEW Main Review

DESCRIPTION

The governance system executes succeeded proposals instantly via the `execute` function in `VetoGovernor`, without a queued state or timelock delay. Once a proposal reaches `ProposalState.Succeeded` (quorum met and `For > Against`), anyone can call `execute` to perform the actions, as the check only verifies status:

```
``solidity
function execute(...) {
  ProposalState status = state(proposalId);
  require(status == ProposalState.Succeeded, "Governor: proposal not successful");
  _proposals[proposalId].executed = true;
  // Proceeds to _execute with target calls
}
```

With a short voting delay (15 minutes) and period (1 week), this allows immediate invocation of sensitive operations like parameter changes or fund transfers post-voting.

RECOMMENDATION

Consider integrating a timelock extension (e.g., inherit from OZ's `GovernorTimelockControl`) with a minimum 48-hour delay between success and execution, adding a queued state for veto or community response.

RESOLUTION

Status: Pending.

INFORMATIONAL I-21

PENDING

Setters Lack Event Emissions

CATEGORY Best Practices
LOCATION ProtocolGovernor.sol
REVIEW Main Review

— DESCRIPTION

Several setter functions in `ProtocolGovernor`, such as `setProposalNumerator`, `setVetoer`, `acceptVetoer` and `renounceVetoer`, update critical state variables without emitting events.

This silently changes proposal thresholds, vetoer roles, or renounces veto power, making it hard for offchain tools, monitors or users to track alterations.

— RECOMMENDATION

Add custom events (e.g., `emit ProposalNumeratorUpdated(old, new);`) to all setters for offchain indexing and transparency.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-22

PENDING

LOCKED and MANAGED Delegate Differences

CATEGORY	Informational
LOCATION	GovernorSimpleVotes.sol: 46-53
REVIEW	Main Review

— DESCRIPTION

In Velodrome the `GovernorSimpleVotes._getVotes()` function assigns the initial weight deposited + earned rewards towards the voting power of the `LOCKED` escrows if the managed lock they are depositing into doesn't have a delegatee. To not double count votes, the team has also disabled `MANAGED` escrow from voting.

In Purrrswap's case, it's possible to vote with all three escrow types. This means the voting power of the `LOCKED` escrow types will always be equal to whatever is delegated to them, not counting their initial deposited weight and rewards earned.

Furthermore, it also means if a managed lock holds a large amount of voting power, the owner may be able to vote and execute proposals unilaterally, creating centralization risk.

— RECOMMENDATION

1. Users should be warned about losing their voting power once they deposit into a managed lock.
2. Be aware of centralization risks if a managed lock accrues significant voting power

— RESOLUTION

Status: Pending.

INFORMATIONAL I-23

PENDING

Immediate Execution Lacks Timelock Delay

CATEGORY Warning
LOCATION VetoGovernor.sol
REVIEW Main Review

DESCRIPTION

The governance system executes succeeded proposals instantly via the `execute` function in `VetoGovernor`, without a queued state or timelock delay. Once a proposal reaches `ProposalState.Succeeded` (quorum met and `For > Against`), anyone can call `execute` to perform the actions, as the check only verifies status:

```
``solidity
function execute(...) {
  ProposalState status = state(proposalId);
  require(status == ProposalState.Succeeded, "Governor: proposal not successful");
  _proposals[proposalId].executed = true;
  // Proceeds to _execute with target calls
}
```

With a short voting delay (15 minutes) and period (1 week), this allows immediate invocation of sensitive operations like parameter changes or fund transfers post-voting.

RECOMMENDATION

Consider integrating a timelock extension (e.g., inherit from OZ's `GovernorTimelockControl`) with a minimum 48-hour delay between success and execution, adding a queued state for veto or community response.

RESOLUTION

Status: Pending.

INFORMATIONAL I-24

PENDING

Unbounded Reward List In MultiRewards

CATEGORY	DoS
LOCATION	MultiRewards.sol: 250
REVIEW	Main Review

— DESCRIPTION

The `MultiRewards` contract allows the `owner` to add reward tokens via the `addReward` function. However, there is no limit on how many reward tokens can be added, and no mechanism to remove tokens once they are included.

Over time, this could lead to an excessively large reward list. Since the `_updateRewards` function iterates through all reward tokens and is called within multiple critical functions (e.g., `stake`, `withdraw`, and `getReward`), a large list could cause these operations to run out of gas and fail. This introduces a potential denial-of-service risk for users attempting to interact with the contract.

— RECOMMENDATION

Consider introducing a cap on the number of reward tokens that can be added or implementing a mechanism to safely remove obsolete reward tokens from the list to avoid unbounded growth.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-25

PENDING

No Quorum Requirement In EpochGovernor

CATEGORY Best Practices
LOCATION EpochGovernor.sol
REVIEW Main Review

— DESCRIPTION

The EpochGovernor contract currently does not enforce any quorum threshold for proposals. As a result, proposals can be executed with minimal participation, even if only a small number of voters engage. This undermines the legitimacy of governance outcomes, as decisions may not reflect the broader consensus of stakeholders.

— RECOMMENDATION

Consider whether a quorum threshold should be introduced to ensure that proposals require minimum participation before execution.

— RESOLUTION

Status: Pending.

INFORMATIONAL I-26

PENDING

Delayed Fee Distribution From `_claimFees`

CATEGORY	Logical Error
LOCATION	<code>Voter.sol</code> : 477
REVIEW	Main Review

— DESCRIPTION

The `_claimFees` function in the `Gauge` contract is invoked during `notifyRewardAmount`, which itself is restricted to calls from the `Voter` contract. The `notifyRewardAmount` function is triggered within the `Voter` contract's `distribute` function—but only when `_claimable > IGauge(_gauge).left()` and `_claimable > DURATION`.

As a result, there are scenarios where the `PurrrLP` contract may have accrued fees ready for distribution, but `_claimFees` is not called because the outer condition in `distribute` is not satisfied. This leads to delayed fee distribution, potentially missing active staking periods and causing some users to receive reduced or no rewards for the time they were staked.

— RECOMMENDATION

Consider providing a dedicated function to claim and distribute fees more promptly, regardless of emission thresholds.

— RESOLUTION

Status: Pending.

HIGH **H-01**

Disabling managed lock mishandles delegations

CATEGORY	Logical Error
LOCATION	VotingEscrow.sol: 262
REVIEW	Remediation Review

— DESCRIPTION

If the NFT being disabled by `VotingEscrow.setManagedState()` is actively delegating, the code will remove its delegation

```
``solidity
_checkpointDelegator(_tokenId, 0, _ownerOf(_tokenId));
_checkpointDelegatee(currentDelegate, delegatedBalance, false);
`
```

Notice that it calls both `checkpointDelegator()` and `checkpointDelegatee()`. The problem with this approach is that calling `_checkpointDelegator()` will internally call

`checkpointDelegatee()` to decrease the delegation for the current delegatee. Invoking that function a second time will result in the delegatee losing twice the power of the managed NFT.

This behavior will cause accounting mismatch, where the sum of the power of all the delegators doesn't match the sum of the power of all the delegates. In result the delegatee has less influence when voting than they should. Furthermore, it can be exploited by the owner of a malicious managed lock that's to be disabled. They can frontrun the disabling transaction and delegate to a desired lock to significantly reduce its voting power after the transaction is executed.

— RECOMMENDATION

Remove the call to `_checkpointDelegatee()` in `setManagedState()`. For reference, see how `_transferFrom()` calls only `_checkpointDelegator()`.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team2/pull/11>

— RESOLUTION

Status: Reported during the fix review period.

HIGH **H-02**

Withdrawal Blocked for Deactivated NFTs

CATEGORY	DoS
LOCATION	Voter.sol: 338
REVIEW	Remediation Review

— DESCRIPTION

The `_increaseAmountFor` function, invoked by `depositFor`, now prevents calls involving deactivated managed NFTs. This restriction prevents further deposits into these NFTs, as intended. However, it also introduces unintended issues elsewhere in the system.

Firstly, the `RewardsDistributor` contract uses `depositFor` during its `claim` process to allocate earned rewards. Since `_increaseAmountFor` now rejects deactivated NFTs, these tokens are unable to claim rewards earned prior to deactivation.

More critically, the `withdrawManaged` function in the `Voter` contract requires that all pending rewards be claimed before a token can be withdrawn from a managed NFT. This creates a deadlock: deactivated managed NFTs cannot claim rewards, and users cannot withdraw their tokens—effectively locking them in.

— RECOMMENDATION

Allow `_increaseAmountFor` to be called on deactivated managed NFTs only when invoked by the `RewardsDistributor` contract. Alternatively, remove the claim requirement from `withdrawManaged`, and clearly document that users should manually call `claim` on the managed NFT prior to withdrawing, to ensure all rewards are collected.

PROOF OF CONCEPT <https://github.com/GuardianOrg/purrrswap-contracts-purrrswap-aerodrome-team1/blob/394bc17c81c92600d20194b5d74c92c37b0568fc/test/DOSwithdrawManaged.t.sol#L12>

— RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-01**

Tail Emissions Cannot Be Maintained Or Decreased

CATEGORY	Logical Error
LOCATION	GovernorSimple.sol
REVIEW	Remediation Review

— DESCRIPTION

The change to the `execute` function in `GovernorSimple` restricts execution to proposals in the `Succeeded` state only:

```
``plain text
require(
status == ProposalState.Succeeded,
"GovernorSimple: proposal not successful"
);
`
```

This function triggers the `nudge` function in `Minter`, which adjusts tail emissions based on the proposal outcome. However, `nudge` is designed to interpret `Succeeded`, `Defeated`, and `Expired` states as signals to increase, decrease, or maintain emissions respectively.

Restricting execution to only `Succeeded` proposals disables the ability to vote for emission decreases (`Defeated`) or to maintain current levels (`Expired`). This change undermines the governance mechanism and prevents the full range of voter intent from being executed.

— RECOMMENDATION

Revert the condition in `GovernorSimple.execute` to allow execution of proposals in `Defeated` and `Expired` states

— RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-02**

Vote Reset Blocks Reward Claims

CATEGORY	Rewards
LOCATION	Voter.sol: 321
REVIEW	Remediation Review

— DESCRIPTION

The `depositManaged` function calls `_claimAllPendingRewards` to collect rewards from all pools the token has voted on before depositing it into a managed NFT. To do this, it uses the `poolVote` function in the `Voter` contract to retrieve the list of voted pools and their associated reward contracts. However, the token's votes are reset earlier within the `depositManaged`, which clears all previous voting data before the rewards are fetched. As a result, `poolVote` returns an empty list, and no rewards are claimed—even if the token had earned rewards prior to the deposit.

— RECOMMENDATION

Modify the `depositManaged` function in the `Voter` contract to call `depositManaged` in the `VotingEscrow` contract before resetting votes, ensuring the tokens voting data remains intact during reward collection. Alternatively, clearly document that users should manually claim rewards prior to calling `depositManaged`.

— RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-03**

Insufficient Approval Breaks Reward Claims

CATEGORY	Rewards
LOCATION	VotingReward.sol: 23
REVIEW	Remediation Review

— DESCRIPTION

The `depositManaged` function attempts to claim all pending rewards for a token before depositing it into a managed NFT. It does so by calling `_claimAllPendingRewards`, which retrieves the token's voted pools via the `poolVote` function in the `Voter` contract, then attempts to claim rewards from the associated reward contracts.

However, the reward contracts restrict `getReward` calls to only the token owner or an approved address. Since `getReward` is called from within the `VotingEscrow` contract—which is neither the owner nor an approved operator—these reward claims consistently fail. This results in expected rewards not being collected when a user deposits into a managed NFT.

— RECOMMENDATION

Modify the `getReward` function to allow calls from the `VotingEscrow` contract, similar to how it currently allows calls from the `Voter`. Alternatively, ensure that frontends and documentation clearly instruct users to claim rewards manually before calling `depositManaged`.

— RESOLUTION

Status: Reported during the fix review period.

LOW **L-01**

Gauge Fees Cannot Be Claimed Post-Kill

CATEGORY	Logical Error
LOCATION	Gauge.sol
REVIEW	Remediation Review

— DESCRIPTION

The fix for M-05 correctly ensures that fees are claimed before a gauge is killed. However, killed gauges are likely to still hold user deposits and continue accruing new fees. Due to a lack of post-kill fee-claim path, these fees are permanently inaccessible.

— RECOMMENDATION

Consider allowing anyone to call the `claimFees` function, instead of just the Voter contract.

— RESOLUTION

Status: Reported during the fix review period.

LOW L-02

FreeManagedReward Fails To Revalidate Token Whitelist

CATEGORY	Validation
LOCATION	FreeManagedReward.sol
REVIEW	Remediation Review

DESCRIPTION

In the current implementation of `FreeManagedReward.notifyRewardAmount`, the contract checks `IVoter(voter).isWhitelistedToken(token)` only once, when the token is first added to the rewards list. After that initial pass, any further calls to `notifyRewardAmount` for that token bypass the whitelist check entirely, even if governance has since removed the token from the global whitelist. This “first-time only” pattern is evident in the code:

```
``solidity
function notifyRewardAmount(address token, uint256 amount) external override nonReentrant {
    address sender = _msgSender();
    if (!isReward[token]) {
        if (!IVoter(voter).isWhitelistedToken(token)) revert NotWhitelisted();
        isReward[token] = true;
        rewards.push(token);
    }
    _notifyRewardAmount(sender, token, amount);
}
```

Because the call to `isWhitelistedToken(token)` is nested inside `if (!isReward[token])`, once `isReward[token]` becomes `true` the whitelist guard is never executed again. This allows old or malicious tokens to continue receiving reward notifications even after removal from the approved token list.

RECOMMENDATION

Adopt the same pattern used in `BribeVotingReward.notifyRewardAmount`, where the whitelist test occurs unconditionally at the top of the function. Move the call to `IVoter(voter).isWhitelistedToken(token)` outside of the `isReward[token]` check so that every invocation enforces the on-chain whitelist. For example:

```
```solidity
```

**L-02** FreeManagedReward Fails To Revalidate Token Whitelist**CONTINUED**

```
function notifyRewardAmount(address token, uint256 amount) external override nonReentrant {
 address sender = _msgSender();
 if (!IVoter(voter).isWhitelistedToken(token)) revert NotWhitelisted();

 if (!isReward[token]) {
 isReward[token] = true;
 rewards.push(token);
 }
 _notifyRewardAmount(sender, token, amount);
}
...
```

**— RESOLUTION**

Status: Reported during the fix review period.

LOW **L-03**

## Deactivated NFTs Can Vote in EpochGovernor

CATEGORY	Validation
LOCATION	GovernorSimpleVotes.sol: 46
REVIEW	Remediation Review

### — DESCRIPTION

The `_getVotes` function in the `ProtocolGovernor` contract was updated to restrict deactivated managed NFTs from participating in governance by returning zero voting power. This ensures that only active NFTs can create and vote on proposals.

However, this restriction is not applied in the `EpochGovernor` contract. As a result, deactivated managed NFTs can still create and vote on proposals within `EpochGovernor`.

### — RECOMMENDATION

Review whether the EpochGovernor contract should adopt the same restriction on deactivated managed NFTs to ensure consistent governance behavior across contracts.

### — RESOLUTION

Status: Reported during the fix review period.

## INFORMATIONAL I-01

# Full voting power redistributed to alive gauges

CATEGORY	Informational
LOCATION	Voter.sol
REVIEW	Remediation Review

## — DESCRIPTION

The internal `_poke()` function in `Voter.sol` was modified to not vote for any pools with inexistent or killed gauges. This approach ensures `poke` can be successfully called on a `tokenId` even if one of the gauges it votes for has been killed.

In result, when `poke()` is called on such NFT, its voting power will be fully utilized for the rest of pools. For example:

- User voted 500 voting power in proportions 20/20/60
- Pools receive 100/100/300 weights
- The gauge for the second pool is killed
- Now when `poke()` is called, the new weights for the pools will be 125/375, i.e the full voting power is utilized

## — RECOMMENDATION

Document this behavior.

## — RESOLUTION

Status: Reported during the fix review period.

## INFORMATIONAL I-02

# Redundant Condition Check in setManagedState Function

CATEGORY	Warning
LOCATION	VotingEscrow.sol
REVIEW	Remediation Review

## DESCRIPTION

In the `setManagedState` function, the condition `if (_state && !deactivated[_tokenId])` includes a redundant check for `!deactivated[_tokenId]`. This redundancy arises because the function already reverts earlier if `deactivated[_tokenId] == _state` via `revert SameState()`. When `_state` is `true` (deactivation mode), the revert ensures that `deactivated[_tokenId]` must be `false` for the function to proceed. Therefore, the `!deactivated[_tokenId]` check is always `true` in this branch and serves no purpose.

## RECOMMENDATION

Remove the redundant check to simplify the code.

## RESOLUTION

Status: Reported during the fix review period.

## § 05

# Disclaimer.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian's position is that each company and individual are responsible for their own due diligence and continuous security. Guardian's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis.

Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract's safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

§ 06

# About Guardian.

Guardian is a smart contract security firm. Every engagement is worked by two independent teams competing on the same scope, each with its own tooling and methods, so no single reviewer's blind spot survives.

**\$45B+**DIGITAL ASSETS UNDER  
REVIEW**200+**

ENGAGEMENTS COMPLETED

**500+**CRITICAL & HIGH FINDINGS  
REPORTED

Reach us at [audits@guardianaudits.com](mailto:audits@guardianaudits.com).