Protocol Audit Report

Version 1.0

ThothRenX

June 9, 2025

Protocol Audit Report

ThothRenX

June 9, 2025

Prepared by: ThothRenX Lead Auditor: - ThothRenX

Table of Contents

- [H-5] Reentrancy Vulnerability in `GaugeExtraRewarder::onReward` Function [External Call Before State Update + Fund Drainage]
- [H-5] Reentrancy Vulnerability in `GaugeExtraRewarder::onReward` Function [External Call Before State Update + Fund Drainage]
- [H-6] Integer Overflow in Reward Calculations [Unchecked Arithmetic + Reward Manipulation] in `GaugeExtraRewarder::_pendingReward`
- [H-7] Reentrancy in `GenesisPool::addIncentives`[Unprotected External Token Transfer]
- [H-8] Reentrancy in claimIncentives [Unprotected External Token Transfer] in `GenesisPool::claimIncentives`
- [H-9] Reentrancy in `GenesisPool::depositToken`[Unprotected External Token Transfer]
- [H-10] Reentrancy in `Bribe::getReward` [Unprotected External Token Transfer]

- Medium
  - [M-1] `GenesisPoolAPI::getAllUserRelatedGenesisPools` Function Contains Unbounded Loops Leading to Denial of Service
  - [M-2] Missing Zero Address Validation in `GenesisPool::Constructor` Leading to Permanent Contract Dysfunction
  - [M-3] Precision Loss in Division Operations [Rounding Down + Reward Loss] in `GaugeExtraRewarder::onReward`
  - [M-4] Missing Input Validation [Invalid Parameters + State Corruption] in `GaugeExtraRewarder::onReward`
  - [M-5] Reentrancy in `AutoVotingEscrowManager ::disableAutoVoting` [Unprotected External Token Transfer]
  - [M-6] Unchecked External Call to `Bribe::IAutomatedVotingManager.originalOwner` [Potential Denial of Service]
  - [M-7] Reentrancy in `GenesisPoolManager ::checkAtEpochFlip` [Unprotected External Calls]
  - [M-8] Reentrancy in `GenesisPoolManager::checkBeforeEpochFlip` [Unprotected External Calls]
  - [M-9] Unchecked External Call to `RewardsDistributor ::avm.getOriginalOwner` [Potential Denial of Service]

- Low
  - [L-1] Missing Zero Address Validation in Critical `AlgebraPoolAPIStorage::initialize` Function
  - [L-2] AlgebraPoolAPIStorage::CLPoolAdmin‘- External Call Without Validation in Access

Protocol Summary

The audited smart contract Blackhole form a decentralized protocol for managing token liquidity pools, voting, and reward distribution. The protocol enables the creation and management of genesis pools for token launches, facilitates automated voting through vote-locked tokens, distributes seasonal rewards to players, manages bribe incentives for voting, and oversees epoch-based pool transitions and reward claims. Specifically, it:

- Genesis Pool Management: Allows token owners to create and fund liquidity pools (GenesisPool, GenesisPoolManager) with native and funding tokens, supporting auction-based token allocation and pool lifecycle management (e.g., pre-listing, launch).
- Automated Voting: Enables users to delegate voting power for ve(3,3) tokens via AutoVotingEscrowManager, integrating with strategies for pool selection and vote weight distribution.
- Reward Distribution: Facilitates seasonal player rewards (BlackClaims) and bribe-based incentives (Bribe, RewardsDistributor) for staked tokens, with mechanisms for claiming, staking, and distributing tokens based on voting power and epoch schedules.

Disclaimer

The ThothRenx team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Scope

The **main focus of contest participants should be to ensure the validity of the code delta introduced by the Blackhole team**. The codebase is effectively a fork of ThenaFi, a project that has undergone several audits and whose behaviour is considered to be sound.

Wardens should maximize the efficiency of the time they invest in the contest by assessing the code delta and documentation of changes provided in the links above.

Files in scope

Contract	SLOC	Purpose	Libraries used
contracts/APIHelper/AlgebraPoolAPI.sol	238	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so core/contracts/interfaces/IAlgebraPool. periphery/contracts/interfaces/INonfun
contracts/APIHelper/AlgebraPoolAPIStorage.sol	43	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so core/contracts/interfaces/IAlgebraPool. upgradeable/access/OwnableUpgradea
contracts/APIHelper/BlackholePairAPIV2.sol	582	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so periphery/contracts/interfaces/IQuoter core/contracts/interfaces/IAlgebraPool.

Contract	SLOC	Purpose	Libraries used
contracts/APIHelper/GenesisPoolAPI.sol	137	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so
contracts/APIHelper/RewardAPI.sol	109	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so
contracts/APIHelper/TokenAPI.sol	51	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so
contracts/APIHelper/TradeHelper.sol	124	N/A	
contracts/APIHelper/veNFTAPI.sol	309	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so
contracts/APIHelper/veNFTAPIV1.sol	367	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so
contracts/AVM/AutoVotingEscrow.sol	54	N/A	
contracts/AVM/AutoVotingEscrowManager.sol	157	N/A	@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.so upgradeable/access/OwnableUpgradea upgradeable/security/ReentrancyGuard
contracts/AVM/interfaces/IAutoVotingEscrow.sol	7	N/A	
contracts/AVM/interfaces/IAutoVotingEscrowManager.sol	4	N/A	
contracts/AlgebraCLVe33/GaugeCL.sol	232	N/A	@openzeppelin/contracts/security/Ree core/contracts/interfaces/IAlgebraPool. periphery/contracts/interfaces/INonfun farming/contracts/interfaces/IAlgebraEt farming/contracts/interfaces/IAlgebraEt farming/contracts/interfaces/IFarmingC farming/contracts/base/IncentiveKey.so core/contracts/interfaces/IERC20Minim farming/contracts/libraries/IncentiveId.
contracts/AlgebraCLVe33/GaugeFactoryCL.sol	104	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea farming/contracts/interfaces/IAlgebraEt core/contracts/interfaces/IAlgebraPool. farming/contracts/base/IncentiveKey.so

Contract	SLOC	Purpose	Libraries used
contracts/Black.sol	81	N/A	
contracts/BlackClaims.sol	145	N/A	
contracts/BlackGovernor.sol	78	N/A	@openzeppelin/contracts/governance/
contracts/Bribes.sol	270	N/A	@openzeppelin/contracts/token/ERC20
contracts/CustomPoolDeployer.sol	150	N/A	@cryptoalgebra/integral-periphery/contracts/interfaces/IAlgebra core/contracts/interfaces/IAlgebraPool. core/contracts/interfaces/vault/IAlgebra
contracts/CustomToken.sol	10	N/A	@openzeppelin/contracts/token/ERC20
contracts/Fan.sol	15	N/A	@openzeppelin/contracts/token/ERC20
contracts/FixedAuction.sol	28	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea
contracts/GaugeExtraRewarder.sol	128	N/A	@openzeppelin/contracts/token/ERC20
contracts/GaugeManager.sol	418	N/A	@openzeppelin/contracts-upgradeable/security/ReentrancyGuard upgradeable/access/OwnableUpgradea upgradeable/token/ERC20/utils/SafeER upgradeable/token/ERC20/IERC20Upgr core/contracts/interfaces/IAlgebraPool. core/contracts/interfaces/vault/IAlgebra farming/contracts/interfaces/IAlgebraEt farming/contracts/base/IncentiveKey.so
contracts/GaugeV2.sol	287	N/A	@openzeppelin/contracts/security/Ree
contracts/GenesisPool.sol	328	N/A	@openzeppelin/contracts/token/ERC20 upgradeable/security/ReentrancyGuard
contracts/GenesisPoolManager.sol	244	N/A	@openzeppelin/contracts/access/Owna upgradeable/access/OwnableUpgradea upgradeable/security/ReentrancyGuard
contracts/GlobalRouter.sol	144	N/A	@openzeppelin/contracts/utils/math/S
contracts/MinterUpgradeable.sol	168	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea

Contract	SLOC	Purpose	Libraries used
contracts/Pair.sol	453	N/A	
contracts/PairFees.sol	46	N/A	
contracts/PairGenerator.sol	16	N/A	
contracts/PermissionsRegistry.sol	151	N/A	
contracts/RewardsDistributor.sol	215	N/A	
contracts/RouterV2.sol	520	N/A	@cryptoalgebra/integral-periphery/contracts/interfaces/IQuoterV periphery/contracts/interfaces/ISwapRo
contracts/SetterTopNPoolsStrategy.sol	69	N/A	@openzeppelin/contracts/access/Owna
contracts/SetterVoteWeightStrategy.sol	57	N/A	@openzeppelin/contracts/access/Owna
contracts/Thenian.sol	113	N/A	@openzeppelin/contracts/token/ERC72
contracts/TokenHandler.sol	150	N/A	
contracts/VeArtProxyUpgradeable.sol	36	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea
contracts/VoterV3.sol	185	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea upgradeable/security/ReentrancyGuard upgradeable/token/ERC20/utils/SafeER upgradeable/token/ERC20/IERC20Upgr
contracts/VotingEscrow.sol	803	N/A	@openzeppelin/contracts/token/ERC72
contracts/WAVAX.sol	56	N/A	
contracts/chainlink/AutomationBase.sol	13	N/A	
contracts/chainlink/AutomationCompatible.sol	4	N/A	
contracts/chainlink/AutomationCompatibleInterface.sol	3	N/A	
contracts/chainlink/EpochController.sol	68	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea
contracts/factories/AuctionFactory.sol	55	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea
contracts/factories/BribeFactoryV3.sol	133	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea

Contract	SLOC	Purpose	Libraries used
contracts/factories/GaugeFactory.sol	33	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea
contracts/factories/GenesisPoolFactory.sol	65	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea
contracts/factories/PairFactory.sol	129	N/A	@openzeppelin/contracts-upgradeable/access/OwnableUpgradea
contracts/governance/Governor.sol	349	N/A	@openzeppelin/contracts/utils/math/S EIP712.sol @openzeppelin/contract
contracts/governance/L2Governor.sol	254	N/A	@openzeppelin/contracts/token/ERC72 EIP712.sol @openzeppelin/contract
contracts/governance/L2GovernorCountingSimple.sol	52	N/A	contracts/governance/L2Governor.sol
contracts/governance/L2GovernorVotes.sol	12	N/A	@openzeppelin/contracts/governance/
contracts/governance/L2GovernorVotesQuorumFraction.sol	30	N/A	contracts/governance/L2GovernorVotes
contracts/interfaces/IAlgebraCLFactory.sol	4	N/A	@cryptoalgebra/integral-core/contracts/interfaces/IAlgebraFacto
contracts/interfaces/IAlgebraCustomCommunityVault.sol	4	N/A	@cryptoalgebra/integral-core/contracts/interfaces/vault/IAlgebra
contracts/interfaces/IAlgebraEternalFarmingCustom.sol	5	N/A	@cryptoalgebra/integral-farming/contracts/base/IncentiveKey.so farming/contracts/interfaces/IAlgebraEt
contracts/interfaces/IAlgebraFarmingProxyPluginFactory.sol	3	N/A	
contracts/interfaces/IAlgebraPoolAPIStorage.sol	3	N/A	
contracts/interfaces/IAuction.sol	3	N/A	
contracts/interfaces/IAuctionFactory.sol	3	N/A	
contracts/interfaces/IAutomatedVotingManager.sol	3	N/A	
contracts/interfaces/IBlackGauge.sol	3	N/A	
contracts/interfaces/IBlackGauge/IClaims.sol	8	N/A	
contracts/interfaces/IBlackGovernor.sol	13	N/A	
contracts/interfaces/IBlackHoleVotes.sol	4	N/A	@openzeppelin/contracts/governance/

Contract	SLOC	Purpose	Libraries used
contracts/interfaces/IBlackholePairApiV2.sol	43	N/A	
contracts/interfaces/IBribe.sol	3	N/A	
contracts/interfaces/IBribeAPI.sol	3	N/A	
contracts/interfaces/IBribeDistribution.sol	3	N/A	
contracts/interfaces/IBribeFactory.sol	3	N/A	
contracts/interfaces/IBribeFull.sol	3	N/A	
contracts/interfaces/IDibs.sol	3	N/A	
contracts/interfaces/IERC20.sol	3	N/A	
contracts/interfaces/IGauge.sol	3	N/A	
contracts/interfaces/IGaugeAPI.sol	3	N/A	
contracts/interfaces/IGaugeCL.sol	4	N/A	@cryptoalgebra/integral-farming/contracts/base/IncentiveKey.so
contracts/interfaces/IGaugeDistribution.sol	3	N/A	
contracts/interfaces/IGaugeFactory.sol	3	N/A	
contracts/interfaces/IGaugeFactoryCL.sol	4	N/A	
contracts/interfaces/IGaugeManager.sol	8	N/A	
contracts/interfaces/IGenesisPool.sol	4	N/A	
contracts/interfaces/IGenesisPoolBase.sol	42	N/A	
contracts/interfaces/IGenesisPoolFactory.sol	3	N/A	
contracts/interfaces/IGenesisPoolManager.sol	3	N/A	
contracts/interfaces/IMinter.sol	3	N/A	
contracts/interfaces/IPair.sol	3	N/A	
contracts/interfaces/IPairCallee.sol	3	N/A	
contracts/interfaces/IPairFactory.sol	3	N/A	
contracts/interfaces/IPairGenerator.sol	3	N/A	
contracts/interfaces/IPairInfo.sol	3	N/A	
contracts/interfaces/IPermissionsRegistry.sol	3	N/A	
contracts/interfaces/IRewardsDistributor.sol	3	N/A	

Contract	SLOC	Purpose	Libraries used
contracts/interfaces/IRouter.sol	8	N/A	
contracts/interfaces/ITokenHandler.sol	3	N/A	
contracts/interfaces/ITopNPoolsStrategy.sol	8	N/A	
contracts/interfaces/IUniswapRouterETH.sol	3	N/A	
contracts/interfaces/IUniswapV2Pair.sol	3	N/A	
contracts/interfaces/IVeArtProxy.sol	8	N/A	
contracts/interfaces/IVoteWeightStrategy.sol	4	N/A	
contracts/interfaces/IVoter.sol	3	N/A	
contracts/interfaces/IVotingEscrow.sol	169	N/A	
contracts/interfaces/IWETH.sol	8	N/A	
contracts/interfaces/IWrappedBribeFactory.sol	3	N/A	
contracts/libraries/Base64.sol	41	N/A	
contracts/libraries/BlackTimeLibrary.sol	52	N/A	
contracts/libraries/Math.sol	3	N/A	
contracts/libraries/PoolsAndRewardsLibrary.sol	10	N/A	
contracts/libraries/SignedSafeMath.sol	33	N/A	
contracts/libraries/VoterFactoryLib.sol	63	N/A	
contracts/libraries/VotingBalanceLogic.sol	178	N/A	
contracts/libraries/VotingDelegationLib.sol	174	N/A	
Totals	**10108**		

Roles

The protocol contains several trusted roles inherited from ThenaFi as well as several roles that are meant to be held by on-chain accounts rather than off-chain entities.

The codebase contains a significant degree of configurability and thus centralization; this is a known issue. This list contains off-chain role entities that complement the roles defined in the invariant chapter and is non-exhaustive as the codebase is vast:

Role	Description
Team	- Can propose a new team address in several contracts- Can set the team's rate in the `MinterUpgradeable` contract- Can configure the reward distributor in the `MinterUpgradeable` contract- Can configure the proposal numerator in the `BlackGovernor` implementation
Team Multisig (`blackMultisig`)	- Can manage roles within the `PermissionsRegistry`
Epoch Manager	- Can manually perform upkeeps in the `EpochController` implementation
Gauge Admin	- Can configure gauge rewarders, rewarder pool IDs, distribution contracts, internal bribes, and the genesis manager fo all gauges under a particular factory
CL Gauge Admin	- Can set the internal bribe of a CL gauge- Can set the `dibs` and `dibsPercentage` configurations of the gauge factory
Bribe Factory	- Can execute emergency mechanisms in `Bribes` such as `Bribes::emergencyRecoverERC20` - Can configure the voter, gauge manager, minter, AVM, and owner of a particular bribe
Team Multisig (`blackTeamMultisig`)	- Is configured as the owner of newly deployed `Bribes` and inherits the Bribe Factory capabilities
Emergency Council	- Can start and stop emergency modes across gauge factories
Voter Administrator	- Can set the AVM, permissoins registry, and max voting number on the `VoterV3` implementation

Executive Summary

We spend 400 hours with 1 auditor using foundry, oppenzepelin docs, codeHawks docs, supervised by Cyfrin ## Issues found | Severity | Number of Issues found | | ———- | ———————— | | High | 10 | | Medium | 9 | | Low | 3 | | Info | 1 | | Total | 23 | | ———- | ———————— |

Findings

High

[H-1] `GenesisPoolAPI::_hasClaimbaleForOwner` Incorrect Claimable Status Detection

Description The `GenesisPoolAPI::_hasClaimbaleForOwner` function contains flawed logic for determining claim eligibility, particularly in `PARTIALLY_LAUNCHED` state.

Impact - Users may miss legitimate claims

- Pool owners see incorrect refund status
- Funds may be permanently locked

Flawed logic:

```
else if(poolStatus == PoolStatus.PARTIALLY_LAUNCHED){
    return tokenAllocation.refundableNativeAmount > 0; // Ignores
    ↪  incentives
}
```

Proof of concept

Code

```
// Test Case 1: Token owner loses access to refunds
TokenAllocation memory allocation = TokenAllocation({
    proposedNativeAmount: 1000e18,
    refundableNativeAmount: 0 // No refundable amount set
});

// Pool partially launched - owner should be able to claim original deposit
PoolStatus status = PoolStatus.PARTIALLY_LAUNCHED;
```

```
address owner = 0x123;

// Current function returns FALSE - owner cannot claim
bool canClaim = _hasClaimbaleForOwner(owner, 0, status, owner, allocation,
↪  incentiveInfo);
assert(!canClaim); // This is WRONG - owner should be able to claim

// Test Case 2: Regular user shows claimable during active phases
address user = 0x456;
uint256 deposit = 500e18;
status = PoolStatus.PRE_LISTING; // Active phase

// Current function returns TRUE - suggesting user can claim during active
↪  phase
canClaim = _hasClaimbaleForOwner(user, deposit, status, owner, allocation,
↪  incentiveInfo);
assert(canClaim); // This is WRONG - user shouldn't be able to claim yet
```

Recomended mitigations

Code

```
function _hasClaimableForOwner(
    address _user,
    uint256 userDeposit,
    PoolStatus poolStatus,
    address tokenOwner,
    TokenAllocation memory tokenAllocation,
    TokenIncentiveInfo memory incentiveInfo
) internal pure returns (bool) {

    if(_user == tokenOwner) {
        // Token owner claimable scenarios
        if(poolStatus == PoolStatus.NOT_QUALIFIED) {
            // Pool failed qualification - owner can claim refunds or
            ↪  incentives
            return (tokenAllocation.refundableNativeAmount > 0 ||
                    incentiveInfo.incentivesToken.length > 0);
        }
        else if(poolStatus == PoolStatus.NATIVE_TOKEN_DEPOSITED) {
            // Pool successful - owner can claim proposed tokens or
            ↪  incentives
            return (tokenAllocation.proposedNativeAmount > 0 ||
                    incentiveInfo.incentivesToken.length > 0);
```

```
        }
        else if(poolStatus == PoolStatus.PRE_LISTING ||
                poolStatus == PoolStatus.PRE_LAUNCH ||
                poolStatus == PoolStatus.PRE_LAUNCH_DEPOSIT_DISABLED) {
            // Active phases - owner has ongoing claims
            return true;
        }
        else if(poolStatus == PoolStatus.PARTIALLY_LAUNCHED) {
            // Partial launch - owner can claim refunds OR remaining
            ↪  deposit
            return (tokenAllocation.refundableNativeAmount > 0 ||
                    tokenAllocation.proposedNativeAmount > 0);
        }
        return false;
    }
    else if(userDeposit > 0) {
        // Regular user claimable scenarios
        if(poolStatus == PoolStatus.NOT_QUALIFIED) {
            // Pool failed - users can claim refunds
            return true;
        }
        else if(poolStatus == PoolStatus.PARTIALLY_LAUNCHED) {
            // Partial launch - users might have refunds available
            return true;
        }
        else if(poolStatus == PoolStatus.NATIVE_TOKEN_DEPOSITED) {
            // Pool successful - users can claim their allocated tokens
            return true;
        }
        // Active phases (PRE_LISTING, PRE_LAUNCH, etc.) - no claiming yet
        return false;
    }

    return false;
}
```

[H-2] `RewardAPI::_getNextEpochRewards` Unsafe Token Metadata Calls Leading to Denial-of-Service

Description The `RewardAPI::_getNextEpochRewards` function makes direct `.symbol()` and `.decimals()` calls to arbitrary token contracts without error handling. These calls are performed in a loop processing potentially multiple tokens.

Impact - A single non-compliant token will cause the entire function call to revert

- Permanent denial-of-service for all bribe-related functionality
- Breaks core contract functionality as the system grows

Proof of Code

```
function _getNextEpochRewards(address _bribe) internal view returns(Bribes
↪  memory _rewards){
    // ...
    for(i; i < totTokens; i++){
        _token = IBribeAPI(_bribe).bribeTokens(i);
        _tokens[i] = _token;
        _symbol[i] = IERC20(_token).symbol(); // Unsafe call
        _decimals[i] = IERC20(_token).decimals(); // Unsafe call
        // ...
    }
}
```

Recomended mitigations Implement try/catch error handling:

```
function safeSymbol(address token) internal view returns (string memory) {
    try IERC20(token).symbol() returns (string memory s) {
        return s;
    } catch {
        return "ERR";
    }
}

function safeDecimals(address token) internal view returns (uint) {
    try IERC20(token).decimals() returns (uint8 d) {
        return d;
    } catch {
        return 0;
    }
}
```

[H-3] `RewardAPI::initialize` Lack of Zero Address Checks in Initialization

Description The initialize function lacks zero address validation for critical dependencies.

Impact - Contract deployment with invalid dependencies

- Permanent bricking requiring redeployment

- Loss of initialization capability

Proof of concept

```
function initialize(address _voter, address _gaugeManager) initializer
↪  public {
    // No address validation
    owner = msg.sender;
    voter = IVoter(_voter);
    gaugeManager = IGaugeManager(_gaugeManager);
    // ...
}
```

Recomended mitigations

```
function initialize(address _voter, address _gaugeManager) initializer
↪  public {
    require(_voter != address(0), "Zero voter");
    require(_gaugeManager != address(0), "Zero gauge manager");
    // ... existing logic ...
}
```

[H-4] Reentrancy Vulnerability in `GenesisPool::_launchCompletely` Functions Leading to Token Drainage and State Corruption

Description

The `GenesisPool::_launchCompletely` and `GenesisPool::_launchPartially` functions contain a critical reentrancy vulnerability due to violating the Checks-Effects-Interactions pattern. Both functions make external calls to router and gauge contracts before updating the critical poolStatus state variable. This allows malicious contracts to re-enter the `launch()` function and potentially drain tokens or corrupt contract state.

Impact - Token Drainage: Attackers can trigger multiple liquidity additions in a single transaction, draining the contract's token reserves - State Corruption: Pool status and allocation tracking variables can become inconsistent, breaking core contract functionality - Double Spending: The same allocated tokens could be used multiple times for liquidity provision - Protocol Disruption: Successful attacks could render the genesis pool mechanism unusable

Proof of concept

Code

```
// Malicious router contract
contract MaliciousRouter {
```

```
    GenesisPool public target;
    uint256 public attackCount;

    constructor(address _target) {
        target = GenesisPool(_target);
    }

    function addLiquidity(
        address tokenA,
        address tokenB,
        bool stable,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external returns (uint amountA, uint amountB, uint liquidity) {

        // Re-enter before poolStatus is updated
        if (attackCount < 3) {
            attackCount++;
            target.launch(address(this), block.timestamp + 1000);
        }

        // Return dummy values to continue execution
        return (amountADesired, amountBDesired, 1000);
    }
}

// Attack execution:
// 1. Deploy MaliciousRouter with GenesisPool address
// 2. Call launch() with malicious router address
// 3. Router re-enters launch() multiple times before poolStatus updates
// 4. Multiple liquidity additions drain contract tokens
```

Recomended mitigations

1. Implement Checks-Effects-Interactions Pattern: Update state variables before making external calls

Code

```
function _launchCompletely(address router, uint256 maturityTime) internal {
```

```
    _setPoolStatus(PoolStatus.LAUNCH); // Update state FIRST
    _approveTokens(router);
    _addLiquidityAndDistribute(router,
    ↪  allocationInfo.allocatedNativeAmount,
    ↪  allocationInfo.allocatedFundingAmount, maturityTime);
}

function _launchPartially(address router, uint256 maturityTime) internal {
    _setPoolStatus(PoolStatus.PARTIALLY_LAUNCHED); // Update state FIRST
    _approveTokens(router);
    _addLiquidityAndDistribute(router,
    ↪  allocationInfo.allocatedNativeAmount,
    ↪  allocationInfo.allocatedFundingAmount, maturityTime);
}
```

2. Add Reentrancy Protection: Utilize the already imported ReentrancyGuard

Code

```
function launch(address router, uint256 maturityTime) external onlyManager
↪  nonReentrant {
    if(genesisInfo.maturityTime > 0) {
        maturityTime = genesisInfo.maturityTime;
    }
    if(_eligbleForCompleteLaunch()){
        _launchCompletely(router, maturityTime);
    }else{
        _launchPartially(router, maturityTime);
    }
}
```

[H-5] Reentrancy Vulnerability in `GaugeExtraRewarder::onReward` Function [External Call Before State Update + Fund Drainage]

Description The `onReward()`function executes an external token transfer before updating critical state variables (`user.amount` and `user.rewardDebt`). This violates the checks-effects-interactions pattern and creates a reentrancy vulnerability where malicious contracts can re-enter the function during the safeTransfer call.

Impact

Critical: Attackers can drain all reward tokens from the contract Users can claim rewards multiple times before their state is updated Total loss of funds allocated for reward distribution Contract becomes

insolvent and unable to pay legitimate rewards

Proof of concept

Code

```
// Malicious contract can implement this in receive()/fallback:
contract MaliciousRewarder {
    GaugeExtraRewarder target;
    uint256 attacks = 0;

    function attack() external {
        // Initial call triggers reentrancy
        target.onReward(address(this), address(this), userBalance);
    }

    receive() external payable {
        if (attacks < 10 && target.rewardToken().balanceOf(address(target))
        ↪  > 0) {
            attacks++;
            target.onReward(address(this), address(this), userBalance);
        }
    }
}
```

Recomended mitigations

1. Implement OpenZeppelin's ReentrancyGuard modifier on the onReward() function
2. Follow checks-effects-interactions pattern: update state before external calls
3. Move the state updates (user.amount and user.rewardDebt) before the safeTransfer call

[H-5] Reentrancy Vulnerability in `GaugeExtraRewarder::onReward` Function [External Call Before State Update + Fund Drainage]

Description The `onReward()`function executes an external token transfer before updating critical state variables (`user.amount` and `user.rewardDebt`). This violates the checks-effects-interactions pattern and creates a reentrancy vulnerability where malicious contracts can re-enter the function during the safeTransfer call.

Impact

Critical: Attackers can drain all reward tokens from the contract Users can claim rewards multiple times before their state is updated Total loss of funds allocated for reward distribution Contract becomes insolvent and unable to pay legitimate rewards

Proof of concept

Code

```
// Malicious contract can implement this in receive()/fallback:
contract MaliciousRewarder {
    GaugeExtraRewarder target;
    uint256 attacks = 0;

    function attack() external {
        // Initial call triggers reentrancy
        target.onReward(address(this), address(this), userBalance);
    }

    receive() external payable {
        if (attacks < 10 && target.rewardToken().balanceOf(address(target))
        ↪  > 0) {
            attacks++;
            target.onReward(address(this), address(this), userBalance);
        }
    }
}
```

Recomended mitigations

1. Implement OpenZeppelin's ReentrancyGuard modifier on the onReward() function
2. Follow checks-effects-interactions pattern: update state before external calls
3. Move the state updates (user.amount and user.rewardDebt) before the safeTransfer call

[H-6] Integer Overflow in Reward Calculations [Unchecked Arithmetic + Reward Manipulation] in `GaugeExtraRewarder::_pendingReward`

Description The reward calculation logic performs multiplication operations without proper overflow checks: `uint256 reward = time * (rewardPerSecond) and accRewardPerShare + (reward * ACC_TOKEN_PRECISION / lpSupply)`. Large time periods or reward rates can cause integer overflow, leading to incorrect calculations.

Impact - Incorrect reward distributions to users - Potential for accumulated rewards per share to wrap around to small values - Users may receive significantly more or fewer rewards than intended - Economic manipulation of the reward mechanism

Proof of concept

```
// If rewardPerSecond = 1e18 and time = type(uint256).max / 1e18 + 1
uint256 time = (type(uint256).max / 1e18) + 1;
uint256 rewardPerSecond = 1e18;
uint256 reward = time * rewardPerSecond; // This will overflow and wrap to
↪  a small number
```

Recomended mitigations - Use SafeMath library for all arithmetic operations - Implement maximum bounds checking for time periods and reward rates - Add overflow checks before multiplication operations - Consider using fixed-point arithmetic libraries for better precision

[H-7] Reentrancy in `GenesisPool::addIncentives`[Unprotected External Token Transfer]

Description The `addIncentives` function transfers tokens using `IERC20.safeTransferFrom` without the `nonReentrant` modifier, despite inheriting `ReentrancyGuardUpgradeable`. This allows a malicious token contract to re-enter the function before state updates, potentially inflating `incentives` balances.

Impact - An attacker could repeatedly credit incentives without transferring additional tokens, leading to unauthorized fund withdrawals in claimIncentives. - Financial Loss: Inflated incentives could drain the contract's token reserves.

Proof of concept

Code

```
function addIncentives(address[] calldata _incentivesToken, uint256[]
↪  calldata _incentivesAmount) external {
    ...
    for(i = 0; i < _incentivesCnt; i++){
        _token = _incentivesToken[i];
        _amount = _incentivesAmount[i];
        if(_token != address(0) && _amount > 0 && (_token ==
        ↪  genesisInfo.nativeToken || tokenHandler.isConnector(_token))){
            IERC20(_token).safeTransferFrom(_sender, address(this),
            ↪  _amount); // Vulnerable call
            if(incentives[_token] == 0){
                incentiveTokens.push(_token);
            }
            incentives[_token] += _amount; // State update after call
        }
    }
    ...
}
```

1. A malicious token's transferFrom re-enters addIncentives with the same _incentivesToken and _incentivesAmount.
2. Since incentives[_token] is updated after the transfer, the attacker can credit _amount multiple times for a single transfer.
3. The attacker claims inflated incentives via claimIncentives.

Recomended mitigations

1. Add the nonReentrant modifier to addIncentives.
2. Update incentives[_token] before safeTransferFrom.
3. Follow the Checks-Effects-Interactions pattern.

Code

```
function addIncentives(address[] calldata _incentivesToken, uint256[]
↪  calldata _incentivesAmount) external nonReentrant {
    ...
    for(i = 0; i < _incentivesCnt; i++){
        _token = _incentivesToken[i];
        _amount = _incentivesAmount[i];
        if(_token != address(0) && _amount > 0 && (_token ==
        ↪  genesisInfo.nativeToken || tokenHandler.isConnector(_token))){
            if(incentives[_token] == 0){
                incentiveTokens.push(_token);
            }
            incentives[_token] += _amount; // Update state first
            IERC20(_token).safeTransferFrom(_sender, address(this),
            ↪  _amount); // Transfer last
        }
    }
    ...
}
```

[H-8] Reentrancy in claimIncentives [Unprotected External Token Transfer] in `GenesisPool::claimIncentives`

Description The `claimIncentives` function transfers incentive tokens using `IERC20.safeTransfer` without the `nonReentrant` modifier. A malicious token could re-enter the function, allowing multiple claims of the same incentives before `incentives[_id]` is reset.

Impact - High Severity: An attacker could drain all incentive tokens by claiming them multiple times in a single transaction. - Financial Loss: Unauthorized token withdrawals could deplete the contract's incentive reserves.

Proof of concept

```
function claimIncentives() external {
    ...
    for(i = 0; i < _incentivesCnt; i++){
        _amount = incentives[incentiveTokens[i]];
        incentives[incentiveTokens[i]] = 0; // Update after transfer
        IERC20(incentiveTokens[i]).safeTransfer(msg.sender, _amount); //
        ↪  Vulnerable call
    }
}
```

1. A malicious token re-enters `claimIncentives` during `safeTransfer`.
2. The re-entrant call claims the same _amount before incentives[_id] is set to 0.
3. The attacker receives multiple transfers of the same incentive tokens.

Recomended mitigations 1. Add the `nonReentrant` modifier to claimIncentives. 2. Reset `incentives[_id]` before safeTransfer.

Example Fixed:

```
function claimIncentives() external nonReentrant {
    ...
    for(i = 0; i < _incentivesCnt; i++){
        _amount = incentives[incentiveTokens[i]];
        incentives[incentiveTokens[i]] = 0; // Update state first
        IERC20(incentiveTokens[i]).safeTransfer(msg.sender, _amount); //
        ↪  Transfer last
    }
}
```

[H-9] Reentrancy in `GenesisPool::depositToken`[Unprotected External Token Transfer]

Description The `depositToken` function uses `IERC20.safeTransferFrom` to transfer funding tokens without the `nonReentrant` modifier. A malicious token could re-enter the function, manipulating `userDeposits` and `totalDeposits` before state updates.

Impact - An attacker could inflate their deposit balance, gaining disproportionate native tokens or disrupting allocation calculations. - Pool Disruption: Incorrect totalDeposits could prevent the pool from reaching PRE_LAUNCH or LAUNCH states.

Proof of concept

Code

```
function depositToken(address spender, uint256 amount) external onlyManager
↪  returns (bool) {
    ...
    IERC20(genesisInfo.fundingToken).safeTransferFrom(spender,
    ↪  address(this), _amount); // Vulnerable call
    if(userDeposits[spender] == 0){
        depositers.push(spender);
    }
    userDeposits[spender] = userDeposits[spender] + _amount; // State
↪  update after call
    totalDeposits += _amount;
    ...
}
```

1. A malicious fundingToken re-enters depositToken during safeTransferFrom.
2. The re-entrant call deposits the same _amount again before userDeposits[spender] is updated.
3. The attacker's userDeposits is inflated, granting excessive native tokens via allocationInfo.allocatedNativeAmount.

Recomended mitigations

1. Add the nonReentrant modifier to depositToken.
2. Update userDeposits and totalDeposits before safeTransferFrom.

Example Fixed:

```
function depositToken(address spender, uint256 amount) external onlyManager
↪  nonReentrant returns (bool) {
    ...
    if(userDeposits[spender] == 0){
        depositers.push(spender);
    }
    userDeposits[spender] += _amount; // Update state first
    totalDeposits += _amount;
    IERC20(genesisInfo.fundingToken).safeTransferFrom(spender,
    ↪  address(this), _amount); // Transfer last
    ...
}
```

[H-10] Reentrancy in `Bribe::getReward` [Unprotected External Token Transfer]

Description The `getReward` function transfers reward tokens using `IERC20.safeTransfer` without the `nonReentrant` modifier, despite inheriting `ReentrancyGuard`. A malicious token contract could re-enter the function before `lastEarn` is updated, allowing multiple claims of the same reward for a given `tokenId`.

Impact

- An attacker could drain the contract's reward tokens by repeatedly claiming rewards in a single transaction, leading to significant financial loss.

- Reward System Disruption: Unauthorized claims could deplete rewards, preventing legitimate users from receiving their earned tokens.

Proof of Concept

```
function getReward(uint256 tokenId, address[] memory tokens) external
↪  nonReentrant {
    address _owner = IVotingEscrow(ve).ownerOf(tokenId);
    if(_owner == avm) {
        _owner = IAutomatedVotingManager(avm).originalOwner(tokenId);
    }
    require(msg.sender == gaugeManager, "NA");
    uint256 _length = tokens.length;
    for (uint256 i = 0; i < _length; i++) {
        uint256 _reward = earned(tokenId, tokens[i]);
        lastEarn[tokens[i]][tokenId] = block.timestamp; // State update
↪  after transfer
        if (_reward > 0) {
            IERC20(tokens[i]).safeTransfer(_owner, _reward); // Vulnerable
             ↪  call
        }
    }
}
```

A malicious token contract re-enters getReward during safeTransfer. The re-entrant call claims the same _reward for tokenId and `tokens[i]` before `lastEarn[tokens[i]][tokenId]` is updated.

The attacker receives multiple transfers of the reward, draining the contract's balance for that token.

Recommended Mitigations

- Reorder operations to update lastEarn before safeTransfer. Ensure the `nonReentrant` modifier is correctly applied (note: it is already present but included here for clarity).

- Follow the Checks-Effects-Interactions pattern to update state before external calls.

Example fix:

```
function getReward(uint256 tokenId, address[] memory tokens) external
↪  nonReentrant {
    address _owner = IVotingEscrow(ve).ownerOf(tokenId);
    if(_owner == avm) {
        _owner = IAutomatedVotingManager(avm).originalOwner(tokenId);
    }
    require(msg.sender == gaugeManager, "NA");
    uint256 _length = tokens.length;
    for (uint256 i = 0; i < _length; i++) {
        uint256 _reward = earned(tokenId, tokens[i]);
        if (_reward > 0) {
            lastEarn[tokens[i]][tokenId] = block.timestamp; // Update state
↪  first
            IERC20(tokens[i]).safeTransfer(_owner, _reward); // Transfer
            ↪  last
            emit RewardPaid(_owner, tokens[i], _reward);
        }
    }
}
```

Medium

[M-1] `GenesisPoolAPI::getAllUserRelatedGenesisPools` Function Contains Unbounded Loops Leading to Denial of Service

Description The function `GenesisPoolAPI::getAllUserRelatedGenesisPools` performs two passes: the first to count the number of eligible pools and the second to fill the array. Both passes iterate over all tokens and pools. This doubles the gas cost and is inefficient. Moreover, the loops are unbounded and could run out of gas.

Impact The function might run out of gas even for a moderate number of pools, making it impossible to retrieve the user's related pools.

Proof of concept

Recomended mitigations Use pagination or off-chain indexing.

[M-2] Missing Zero Address Validation in `GenesisPool::Constructor` Leading to Permanent Contract Dysfunction

Description The `GenesisPool constructor` accepts critical addresses (`_genesisManager, _tokenHandler, _tokenOwner, _nativeToken, _fundingToken`) without validating they are not zero addresses. Since these are immutable variables or set only once during deployment, providing zero addresses would permanently brick the contract with no recovery mechanism.

Impact - Permanent Contract Failure: Contract becomes completely unusable if deployed with zero addresses - No Recovery Mechanism: Cannot redeploy or fix the contract once deployed with invalid addresses - Cascading Failures: External calls to zero addresses will always revert, breaking all functionality - Financial Loss: Any tokens sent to a bricked contract would be permanently locked

Proof of concept

Code

```
// Deploying with zero addresses bricks the contract
GenesisPool pool = new GenesisPool(
    address(0),        // genesisManager - all onlyManager functions fail
    validTokenHandler, // tokenHandler works
    address(0),        // tokenOwner - breaks ownership checks
    address(0),        // nativeToken - breaks all token operations
    validFundingToken  // fundingToken works
);

// Any subsequent calls will fail:
pool.setGenesisPoolInfo(...); // Reverts: msg.sender != address(0)
pool.addIncentives(...);      // Reverts: msg.sender != address(0)
// Contract is permanently unusable
```

Recomended mitigations 1. Add Zero Address Validation in Constructor:

Code

```
constructor(
    address _genesisManager,
    address _tokenHandler,
    address _tokenOwner,
    address _nativeToken,
    address _fundingToken
) {
    require(_genesisManager != address(0), "GenesisPool: genesisManager
    ↪  cannot be zero address");
```

```
        require(_tokenHandler != address(0), "GenesisPool: tokenHandler cannot
        ↪  be zero address");
        require(_tokenOwner != address(0), "GenesisPool: tokenOwner cannot be
        ↪  zero address");
        require(_nativeToken != address(0), "GenesisPool: nativeToken cannot be
        ↪  zero address");
        require(_fundingToken != address(0), "GenesisPool: fundingToken cannot
        ↪  be zero address");

        genesisInfo.tokenOwner = _tokenOwner;
        genesisInfo.nativeToken = _nativeToken;
        genesisInfo.fundingToken = _fundingToken;

        genesisManager = _genesisManager;
        tokenHandler = ITokenHandler(_tokenHandler);

        totalDeposits = 0;
        liquidity = 0;
        tokenOwnerUnstaked = 0;
    }
```

[M-3] Precision Loss in Division Operations [Rounding Down + Reward Loss] in `GaugeExtraRewarder::onReward`

Description Multiple division operations in the contract can result in precision loss due to Solidity's integer division rounding down: `amount / distributePeriod`, `userBalance * accRewardPerShare / ACC_TOKEN_PRECISION`, and `reward * ACC_TOKEN_PRECISION / lpSupply`.

Impact - Users consistently receive fewer rewards than mathematically correct - Accumulated precision loss over time leads to significant reward shortfall - Unfair distribution favoring the protocol over users - Dust amounts become unclaimable

Proof of concept

```
// Example: If amount = 999 and distributePeriod = 1000
uint256 amount = 999;
uint256 distributePeriod = 1000;
uint256 _rewardPerSecond = amount / distributePeriod; // Results in 0,
↪  losing 999 tokens
//
```

Recomended mitigations - Implement proper rounding mechanisms (round half up instead of down) -

Use higher precision constants (e.g., 1e18 instead of 1e12) - Add minimum reward thresholds to prevent zero rewards - Consider accumulating remainder values for future distribution

[M-4] Missing Input Validation [Invalid Parameters + State Corruption] in `GaugeExtraRewarder::onReward`

Description Critical functions lack proper input validation. The `onReward()` function doesn't validate `userBalance`, `setDistributionRate()` doesn't check for amount > 0, and the constructor doesn't validate the gauge address.

Impact

- Invalid state transitions can corrupt contract state
- Zero or negative values can break reward calculations
- Unvalidated addresses can cause external call failures
- Edge cases may result in unexpected behavior

Proof of concept

```
// Calling setDistributionRate with 0 amount
setDistributionRate(0); // This sets rewardPerSecond to 0, stopping all
↪  rewards

// Calling onReward with max uint256
onReward(user, recipient, type(uint256).max); // Could cause overflow in
↪  debt calculations
```

Recomended mitigations - Implement timelock mechanisms for critical parameter changes - Use multi-signature wallets for owner functions - Add emergency pause functionality with time limits - Emit events and provide advance notice for critical changes ## [M-5] Reentrancy in `AutoVotingEscrowManager::disableAutoVoting` [Unprotected External Token Transfer]

Description The `disableAutoVoting` function transfers tokens using `votingEscrow.transferFrom` without the `nonReentrant` modifier, despite inheriting `ReentrancyGuardUpgradeable`. A malicious token contract or `votingEscrow` implementation could re-enter the function, manipulating `tokenIdToAVMId` or `nextAvailableAVMIndex` before state updates, potentially allowing unauthorized access or disrupting Auto Voting Manager (AVM) tracking.

Impact - An attacker could re-enter `disableAutoVoting`, potentially reusing a `tokenId` or corrupting the avms array, leading to incorrect ownership tracking or denial of service for legitimate users.

- Pool Disruption: Incorrect state updates could prevent users from disabling auto-voting or cause the contract to misassign AVM indices, affecting voting power calculations.

Proof of concept

Code

```
function disableAutoVoting(uint256 tokenId) external {
    uint256 avmIdxOneBased = tokenIdToAVMId[tokenId];
    require(avmIdxOneBased > 0 && avmIdxOneBased - 1 < avms.length, "out of
    ↪  range");
    require(!BlackTimeLibrary.isLastHour(block.timestamp), "LH");
    IAutoVotingEscrow avm = avms[avmIdxOneBased - 1];
    require(avm.lockOwner(tokenId) == msg.sender, "Not owner");

    avm.releaseLock(tokenId);
    IVoter(voter).reset(tokenId);
    votingEscrow.transferFrom(address(avm), msg.sender, tokenId); //
↪  Vulnerable call
    delete tokenIdToAVMId[tokenId]; // State update after call

    if ((!avm.isFull()) && (avmIdxOneBased - 1 < nextAvailableAVMIndex)) {
        nextAvailableAVMIndex = avmIdxOneBased - 1;
    }
}
```

1. A malicious `votingEscrow` contract re-enters `disableAutoVoting` during transferFrom.
2. The re-entrant call attempts to disable the same tokenId again before `tokenIdToAVMId[tokenId]` is deleted.
3. This could allow multiple state updates or corrupt `nextAvailableAVMIndex`, enabling an attacker to manipulate AVM assignments or disrupt user withdrawals.

Recomended mitigations

1. Add the nonReentrant modifier to disableAutoVoting.
2. Update state (tokenIdToAVMId, nextAvailableAVMIndex) before the transferFrom call.
3. Follow the Checks-Effects-Interactions pattern to ensure state changes occur before external calls.

Code

```
function disableAutoVoting(uint256 tokenId) external nonReentrant {
    uint256 avmIdxOneBased = tokenIdToAVMId[tokenId];
    require(avmIdxOneBased > 0 && avmIdxOneBased - 1 < avms.length, "out of
    ↪  range");
```

```
    require(!BlackTimeLibrary.isLastHour(block.timestamp), "LH");
    IAutoVotingEscrow avm = avms[avmIdxOneBased - 1];
    require(avm.lockOwner(tokenId) == msg.sender, "Not owner");

    avm.releaseLock(tokenId);
    IVoter(voter).reset(tokenId);
    delete tokenIdToAVMId[tokenId]; // Update state first
    if ((!avm.isFull()) && (avmIdxOneBased - 1 < nextAvailableAVMIndex)) {
        nextAvailableAVMIndex = avmIdxOneBased - 1;
    }
    votingEscrow.transferFrom(address(avm), msg.sender, tokenId); //
↪   Transfer last
}
```

[M-2] Unchecked External Calls to `AutoVotingEscrowManager::AutoVotingEscrow` [Potential Denial of Service]

Description The `disableAutoVoting` function makes external calls to `avm.releaseLock` and `IVoter(voter).reset` without handling potential reverts. A malicious or faulty `AutoVotingEscrow` or `voter` contract could revert, preventing users from disabling auto-voting and locking tokens in the contract.

Impact - Users may be unable to withdraw their tokens, leading to a denial of service and loss of access to locked assets. - User Trust: Persistent failures could deter users from participating in the auto-voting system, impacting the protocol's adoption.

Proof of Concept

```
function disableAutoVoting(uint256 tokenId) external {
    ...
    IAutoVotingEscrow avm = avms[avmIdxOneBased - 1];
    require(avm.lockOwner(tokenId) == msg.sender, "Not owner");

    avm.releaseLock(tokenId); // Vulnerable call
    IVoter(voter).reset(tokenId); // Vulnerable call
    votingEscrow.transferFrom(address(avm), msg.sender, tokenId);
    ...
}
```

1. A malicious `avm` contract reverts on `releaseLock`, or `voter` reverts on `reset`.
2. The disableAutoVoting transaction fails, preventing the user from retrieving their tokenId.
3. Users are stuck with tokens locked in the AVM, unable to disable auto-voting.

Recommended Mitigations

1. Wrap external calls to avm.releaseLock and IVoter(voter).reset in try-catch blocks to handle reverts gracefully.
2. Implement a fallback mechanism to allow partial execution (e.g., skip reset if it fails but proceed with token transfer).
3. Validate avm and voter contract addresses during initialization or updates to ensure trustworthiness.

Example fix:

```
function disableAutoVoting(uint256 tokenId) external nonReentrant {
    uint256 avmIdxOneBased = tokenIdToAVMId[tokenId];
    require(avmIdxOneBased > 0 && avmIdxOneBased - 1 < avms.length, "out of
    ↪  range");
    require(!BlackTimeLibrary.isLastHour(block.timestamp), "LH");
    IAutoVotingEscrow avm = avms[avmIdxOneBased - 1];
    require(avm.lockOwner(tokenId) == msg.sender, "Not owner");

    try avm.releaseLock(tokenId) {} catch {
        emit ReleaseLockFailed(tokenId); // Log failure but continue
    }
    try IVoter(voter).reset(tokenId) {} catch {
        emit ResetFailed(tokenId); // Log failure but continue
    }
    delete tokenIdToAVMId[tokenId];
    if ((!avm.isFull()) && (avmIdxOneBased - 1 < nextAvailableAVMIndex)) {
        nextAvailableAVMIndex = avmIdxOneBased - 1;
    }
    votingEscrow.transferFrom(address(avm), msg.sender, tokenId);
}
```

[H-1] Reentrancy in `BlackClaims::revokeUnclaimedReward` [Unprotected External Token Transfer]

Description The `revokeUnclaimedReward` function transfers tokens to the treasury using `token.transfer` without the `nonReentrant` modifier. A malicious token contract could re-enter the function before the `remaining_reward_amount` is updated, allowing multiple transfers of the same reward amount.

Impact - An attacker could drain the contract's token balance by repeatedly transferring remaining_reward_amount before it is reduced. - Financial Loss: Unauthorized token withdrawals could deplete the contract's reward reserves, affecting legitimate users' ability to claim rewards.

Proof of Concept

```
function revokeUnclaimedReward() external onlyOwner {
    Season storage _season = season;
    uint256 _remaining_reward_amount = _season.remaining_reward_amount;
    require(_season.start_time > 0, "SEASON NOT FOUND");
    require(isSeasonClaimingEnded(), "SEASON_CLAIM_NOT_ENDED");
    require(_remaining_reward_amount > 0, "ZERO_REMAINING_AMOUNT");

    bool transfer_success = token.transfer(treasury,
↪   _remaining_reward_amount); // Vulnerable call
    require(transfer_success, "FAILED TRANSFER");
    _season.remaining_reward_amount -= uint128(_remaining_reward_amount);
↪   // State update after call
}
```

1. A malicious token contract re-enters revokeUnclaimedReward during token.transfer.
2. The re-entrant call triggers another transfer of `_remaining_reward_amount` before `_season.remaining_reward_amount` is updated.
3. The attacker receives multiple transfers of the same reward amount, draining the contract.

Recommended Mitigations 1. Implement the ReentrancyGuard from OpenZeppelin and add the nonReentrant modifier to revokeUnclaimedReward.

2. Update _season.remaining_reward_amount before the token.transfer call.
3. Follow the Checks-Effects-Interactions pattern to ensure state changes occur before external calls.

Example fix:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract BlackClaims is IBlackClaims, ReentrancyGuard {
    ...
    function revokeUnclaimedReward() external onlyOwner nonReentrant {
        Season storage _season = season;
        uint256 _remaining_reward_amount = _season.remaining_reward_amount;
        require(_season.start_time > 0, "SEASON NOT FOUND");
        require(isSeasonClaimingEnded(), "SEASON_CLAIM_NOT_ENDED");
        require(_remaining_reward_amount > 0, "ZERO_REMAINING_AMOUNT");

        _season.remaining_reward_amount -=
↪   uint128(_remaining_reward_amount); // Update state first
        bool transfer_success = token.transfer(treasury,
↪   _remaining_reward_amount); // Transfer last
```

```
        require(transfer_success, "FAILED TRANSFER");
    }
    ...
}
```

[H-2] Reentrancy in `BlackClaims::finalize` [Unprotected External Token Transfer]

Description The `finalize` function uses `token.transferFrom` to pull tokens from the treasury without the `nonReentrant` modifier. A malicious token contract could re-enter the function, potentially manipulating `_season.remaining_reward_amount` or `_season.claim_end_time`before state updates.

Impact - High Severity: An attacker could repeatedly trigger token transfers or disrupt season finalization, leading to incorrect reward allocations or unauthorized token movements.

- Pool Disruption: Reentrancy could cause inconsistent state, preventing proper season finalization and affecting reward claims.

Proof of Concept

```
function finalize(uint256 claim_duration_) external onlyOwner {
    Season storage _season = season;
    require(_season.start_time > 0, "SEASON NOT FOUND");
    require(!isSeasonFinalized(), "SEASON_FINALIZED");
    require(_season.reward_amount > 0, "NO REWARD AMOUNT");
    require(claim_duration_ >= 1 days && claim_duration_ < 1000 days,
    ↪  "CLAIM DURATION OUT OF BOUNDS");

    bool transfer_success = token.transferFrom(treasury, address(this),
↪  _season.reward_amount); // Vulnerable call
    require(transfer_success, "FAILED TRANSFER");

    _season.remaining_reward_amount = _season.reward_amount; // State
↪  update after call
    _season.claim_end_time = block.timestamp + claim_duration_;
}
```

1. A malicious token contract re-enters finalize during token.transferFrom.
2. The re-entrant call attempts to finalize the season again, potentially overwriting _season.remaining_reward_amount or _season.claim_end_time. 3 This could lead to inflated reward amounts or incorrect claim periods, disrupting the reward distribution process.

Recommended Mitigations

1. Add the ReentrancyGuard and apply the nonReentrant modifier to finalize.

2. Update _season.remaining_reward_amount and _season.claim_end_time before the token.transferFrom call.
3. Ensure state changes precede external calls.

Example fix:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract BlackClaims is IBlackClaims, ReentrancyGuard {
    ...
    function finalize(uint256 claim_duration_) external onlyOwner
    ↪  nonReentrant {
        Season storage _season = season;
        require(_season.start_time > 0, "SEASON NOT FOUND");
        require(!isSeasonFinalized(), "SEASON_FINALIZED");
        require(_season.reward_amount > 0, "NO REWARD AMOUNT");
        require(claim_duration_ >= 1 days && claim_duration_ < 1000 days,
        ↪  "CLAIM DURATION OUT OF BOUNDS");

        _season.remaining_reward_amount = _season.reward_amount; // Update
↪  state first
        _season.claim_end_time = block.timestamp + claim_duration_;
        bool transfer_success = token.transferFrom(treasury, address(this),
↪  _season.reward_amount); // Transfer last
        require(transfer_success, "FAILED TRANSFER");
    }
    ...
}
```

[M-3] Unchecked External Call to `BlackClaims::_ve.create_lock_for` in `BlackClaims::claimAndStakeReward` [Potential Denial of Service]

Description The `claimAndStakeReward` function calls `_ve.create_lock_for` without handling potential reverts. A malicious or faulty IVotingEscrow contract could revert, preventing users from staking their rewards and potentially locking funds in the contract.

Impact - Users may be unable to stake their rewards, leading to a denial of service and reduced functionality of the reward system. - User Trust: Persistent failures could discourage participation in the rewards program, impacting protocol adoption.

Proof of Concept

```
function claimAndStakeReward(uint percent) external returns (uint) {
    ...
```

```
    token.approve(address(_ve), staked_reward);
    uint _tokenId = _ve.create_lock_for(staked_reward, MAX_PERIOD,
↪   msg.sender, true); // Vulnerable call
    emit StakedRewards(msg.sender, staked_reward);
    return _tokenId;
}
```

1. A malicious _ve contract reverts on create_lock_for.
2. The claimAndStakeReward transaction fails, preventing the user from staking their reward.
3. Users are unable to complete the staking process, limiting access to the ve(3,3) system benefits.

Recommended Mitigations - Wrap `_ve.create_lock_for` in a `try-catch` block to handle reverts gracefully.

- Implement a `fallback` mechanism to allow users to claim rewards without staking if `_ve.create_lock_for` fails.
- Validate the _ve contract address during initialization to ensure trustworthiness.

Example fix:

```
function claimAndStakeReward(uint percent) external returns (uint) {
    ...
    token.approve(address(_ve), staked_reward);
    uint _tokenId;
    try _ve.create_lock_for(staked_reward, MAX_PERIOD, msg.sender, true)
    ↪   returns (uint tokenId) {
        _tokenId = tokenId;
        emit StakedRewards(msg.sender, staked_reward);
    } catch {
        // Fallback: Transfer reward to user without staking
        bool transfer_success = token.transfer(msg.sender, staked_reward);
        require(transfer_success, "FAILED TRANSFER");
        emit StakedRewards(msg.sender, 0); // Log failed staking attempt
    }
    return _tokenId;
}
```

[M-6] Unchecked External Call to `Bribe::IAutomatedVotingManager.originalOwner` [Potential Denial of Service]

Description

The `getReward` function calls `IAutomatedVotingManager(avm).originalOwner(tokenId)` without handling potential reverts. A malicious or faulty avm contract could revert, preventing users from claiming rewards and causing a denial of service.

Impact

- Users may be unable to claim their rewards, disrupting the bribe system's functionality and reducing user trust.

- Operational Disruption: Persistent failures could lock rewards in the contract, affecting the protocol's reward distribution.

Proof of Concept

```
function getReward(uint256 tokenId, address[] memory tokens) external
↪  nonReentrant {
    address _owner = IVotingEscrow(ve).ownerOf(tokenId);
    if(_owner == avm) {
        _owner = IAutomatedVotingManager(avm).originalOwner(tokenId); //
↪  Vulnerable call
    }
    ...
}
```

- A malicious avmcontract reverts on `originalOwner(tokenId)`. The `getReward` transaction fails, preventing the user from claiming rewards for `tokenId`.
- Users are unable to access their earned rewards, leading to a denial of service.

Recommended Mitigations

- Wrap the `originalOwner` call in a `try-catch` block to handle reverts gracefully.
- Implement a fallback mechanism (e.g., use IVotingEscrow(ve).ownerOf(tokenId) if originalOwner fails). Validate the avm contract address during initialization or updates to ensure trustworthiness.

Example fix:

```
function getReward(uint256 tokenId, address[] memory tokens) external
↪  nonReentrant {
    address _owner = IVotingEscrow(ve).ownerOf(tokenId);
    if(_owner == avm) {
        try IAutomatedVotingManager(avm).originalOwner(tokenId) returns
        ↪  (address originalOwner) {
```

```
            _owner = originalOwner;
        } catch {
            // Fallback to ve owner if avm call fails
            emit OriginalOwnerCallFailed(tokenId);
        }
    }
    require(msg.sender == gaugeManager, "NA");
    uint256 _length = tokens.length;
    for (uint256 i = 0; i < _length; i++) {
        uint256 _reward = earned(tokenId, tokens[i]);
        if (_reward > 0) {
            lastEarn[tokens[i]][tokenId] = block.timestamp;
            IERC20(tokens[i]).safeTransfer(_owner, _reward);
            emit RewardPaid(_owner, tokens[i], _reward);
        }
    }
}
```

[M-7] Reentrancy in `GenesisPoolManager ::checkAtEpochFlip` [Unprotected External Calls]

Description The checkAtEpochFlip function makes external calls to `IGenesisPool(_genesisPool).eligbleF tokenHandler.whitelistToken()`, and `IGenesisPool(_genesisPool).launch()` without the nonReentrant modifier, despite inheriting `ReentrancyGuardUpgradeable`. A malicious genesisPool or tokenHandler contract could re-enter the function, potentially manipulating the state of liveNativeTokens or triggering unintended pool launches.

Impact

- An attacker could disrupt the epoch flip process, causing incorrect pool state transitions or unauthorized token whitelisting, leading to financial loss or protocol instability.

- Pool Disruption: Reentrancy could result in duplicate launches, incorrect pool statuses, or removal of valid tokens from liveNativeTokens.

Proof of Concept

```
function checkAtEpochFlip() external {
    require(epochController == msg.sender, "NA");
    uint256 _proposedTokensCnt = liveNativeTokens.length;
    uint256 i;
    address _genesisPool;
    PoolStatus _poolStatus;
```

```
    address nativeToken;

    for(i = _proposedTokensCnt; i > 0; i--){
        nativeToken = liveNativeTokens[i-1];
        _genesisPool = genesisFactory.getGenesisPool(nativeToken);
        _poolStatus = IGenesisPool(_genesisPool).poolStatus();

        if(_poolStatus == PoolStatus.PRE_LISTING &&
        ↪  IGenesisPool(_genesisPool).eligbleForPreLaunchPool()){ //
        ↪  Vulnerable call
            tokenHandler.whitelistToken(nativeToken); // Vulnerable call
            _preLaunchPool(_genesisPool);
        }else if(_poolStatus == PoolStatus.PRE_LAUNCH_DEPOSIT_DISABLED){
            _launchPool(nativeToken, _genesisPool); // Calls launch()
        }
    }
}
```

- A malicious genesisPool contract re-enters checkAtEpochFlip during eligbleForPreLaunchPool() or launch(). The re-entrant call manipulates liveNativeTokens by triggering additional iterations or modifying pool states before the loop completes.
- This could lead to duplicate token whitelisting, premature pool launches, or removal of valid tokens from liveNativeTokens.

Recommended Mitigations

- Add the nonReentrant modifier to checkAtEpochFlip.
- Use a local array to cache liveNativeTokens before processing to prevent state changes during iteration. Wrap external calls in try-catch blocks to handle potential reverts gracefully.

Example fix:

```
function checkAtEpochFlip() external nonReentrant {
    require(epochController == msg.sender, "NA");
    address[] memory tokens = liveNativeTokens; // Cache tokens
    uint256 _proposedTokensCnt = tokens.length;
    uint256 i;
    address _genesisPool;
    PoolStatus _poolStatus;
    address nativeToken;

    for(i = _proposedTokensCnt; i > 0; i--){
        nativeToken = tokens[i-1];
```

```
        _genesisPool = genesisFactory.getGenesisPool(nativeToken);
        _poolStatus = IGenesisPool(_genesisPool).poolStatus();

        if(_poolStatus == PoolStatus.PRE_LISTING) {
            bool eligible = false;
            try IGenesisPool(_genesisPool).eligbleForPreLaunchPool()
            ↪  returns (bool result) {
                eligible = result;
            } catch {}
            if(eligible){
                try tokenHandler.whitelistToken(nativeToken) {} catch {}
                _preLaunchPool(_genesisPool);
            }
        } else if(_poolStatus == PoolStatus.PRE_LAUNCH_DEPOSIT_DISABLED){
            _launchPool(nativeToken, _genesisPool);
        }
    }
}
```

[M-8] Reentrancy in `GenesisPoolManager::checkBeforeEpochFlip` [Unprotected External Calls]

Description The `checkBeforeEpochFlip` function makes external calls to `IGenesisPool(_genesisPool).eligbleForDisqualify()` and `IGenesisPool(_genesisPool).setPoolSt` without the `nonReentrant` modifier. A malicious `genesisPool` contract could re-enter the function, manipulating `liveNativeTokens` or pool statuses during the epoch flip preparation.

Impact

- Reentrancy could lead to incorrect pool disqualifications, unauthorized state changes, or removal of valid tokens from liveNativeTokens, causing financial or operational disruptions.
- Protocol Instability: Inconsistent pool states could prevent proper epoch transitions, affecting user deposits and pool launches.

Proof of Concept

```
function checkBeforeEpochFlip() external {
    require(epochController == msg.sender, "NA");
    uint _period = pre_epoch_period;
    if (block.timestamp >= _period + WEEK) {
        uint256 _proposedTokensCnt = liveNativeTokens.length;
        uint256 i;
```

```
        address _genesisPool;
        PoolStatus _poolStatus;
        address nativeToken;

        for(i = _proposedTokensCnt; i > 0; i--){
            nativeToken = liveNativeTokens[i-1];
            _genesisPool = genesisFactory.getGenesisPool(nativeToken);
            _poolStatus = IGenesisPool(_genesisPool).poolStatus();

            if(_poolStatus == PoolStatus.PRE_LISTING &&
            ↪  IGenesisPool(_genesisPool).eligbleForDisqualify()){ //
            ↪  Vulnerable call
                pairFac-
↪   tory.setGenesisStatus(IGenesisPool(_genesisPool).getLiquidityPoolInfo().pairAddress,
↪   false);
                IGenesis-
                ↪  Pool(_genesisPool).setPoolStatus(PoolStatus.NOT_QUALIFIED);
                ↪  // Vulnerable call
                _removeLiveToken(nativeToken);
            }
            else if(_poolStatus == PoolStatus.PRE_LAUNCH){
                IGenesis-
                ↪  Pool(_genesisPool).setPoolStatus(PoolStatus.PRE_LAUNCH_DEPOSIT_DISAB
            }
        }
        pre_epoch_period = BlackTimeLibrary.currPreEpoch(block.timestamp);
    }
}
```

- A malicious `genesisPool` re-enters `checkBeforeEpochFlip` during `eligbleForDisqualify()` or `setPoolStatus()`.
- The re-entrant call manipulates liveNativeTokens or triggers additional disqualifications before the loop completes.
- This could lead to incorrect token removals or pool state transitions, disrupting the epoch flip process.

Recommended Mitigations

- Add the nonReentrant modifier to checkBeforeEpochFlip. Cache liveNativeTokens in a local array before iteration to prevent state changes.
- Wrap external calls in try-catch blocks to handle reverts.

Example fix:

```
function checkBeforeEpochFlip() external nonReentrant {
    require(epochController == msg.sender, "NA");
    uint _period = pre_epoch_period;
    if (block.timestamp >= _period + WEEK) {
        address[] memory tokens = liveNativeTokens; // Cache tokens
        uint256 _proposedTokensCnt = tokens.length;
        uint256 i;
        address _genesisPool;
        PoolStatus _poolStatus;
        address nativeToken;

        for(i = _proposedTokensCnt; i > 0; i--){
            nativeToken = tokens[i-1];
            _genesisPool = genesisFactory.getGenesisPool(nativeToken);
            _poolStatus = IGenesisPool(_genesisPool).poolStatus();

            if(_poolStatus == PoolStatus.PRE_LISTING) {
                bool disqualify = false;
                try IGenesisPool(_genesisPool).eligbleForDisqualify()
                ↪  returns (bool result) {
                    disqualify = result;
                } catch {}
                if(disqualify){
                    pairFac-
↪   tory.setGenesisStatus(IGenesisPool(_genesisPool).getLiquidityPoolInfo().pairAddress,
↪   false);
                    try IGenesis-
                    ↪  Pool(_genesisPool).setPoolStatus(PoolStatus.NOT_QUALIFIED)
                    ↪  {} catch {}
                    _removeLiveToken(nativeToken);
                }
            }
            else if(_poolStatus == PoolStatus.PRE_LAUNCH){
                try IGenesis-
                ↪  Pool(_genesisPool).setPoolStatus(PoolStatus.PRE_LAUNCH_DEPOSIT_DISAB
                ↪  {} catch {}
            }
        }
        pre_epoch_period = BlackTimeLibrary.currPreEpoch(block.timestamp);
    }
}
```

[H-1] Reentrancy in `RewardsDistributor::claim_many`[Unprotected External Token Trans-

fer]

Description

The `claim_many` function transfers tokens using `IERC20(token).transfer` for expired non-permanent locks without the nonReentrant modifier. A malicious token contract could re-enter the function, allowing multiple claims of rewards for the same tokenId before token_last_balance or time_cursor_of is updated, leading to unauthorized token withdrawals.

Impact

High Severity: An attacker could drain the contract's token balance by repeatedly claiming rewards for multiple tokenIds in a single transaction, causing significant financial loss.

Reward System Disruption: Unauthorized claims could deplete available rewards, preventing legitimate users from claiming their earned tokens.

Proof of Concept

```
function claim_many(uint[] memory _tokenIds) external returns (bool) {
    uint _last_token_time = last_token_time;
    _last_token_time = _last_token_time / WEEK * WEEK;
    address _voting_escrow = voting_escrow;
    uint total = 0;

    for (uint i = 0; i < _tokenIds.length; i++) {
        uint _tokenId = _tokenIds[i];
        if (_tokenId == 0) break;
        uint amount = _claim(_tokenId, _voting_escrow, _last_token_time);
        if (amount != 0) {
            IVotingEscrow.LockedBalance memory _locked =
↪   IVotingEscrow(_voting_escrow).locked(_tokenId);
            if(_locked.end < block.timestamp && !_locked.isPermanent){
                address _nftOwner =
↪   IVotingEscrow(_voting_escrow).ownerOf(_tokenId);
                IERC20(token).transfer(_nftOwner, amount); // Vulnerable
                ↪  call
            } else {
                IVotingEscrow(_voting_escrow).deposit_for(_tokenId,
                ↪  amount);
            }
            total += amount;
        }
    }
```

```
    if (total != 0) {
        token_last_balance -= total; // State update after transfer
    }
    return true;
}
```

- A malicious token contract re-enters claim_many during `IERC20(token).transfer` for an expired non-permanent lock.

The re-entrant call claims rewards for the same or different tokenIds before token_last_balance is updated. The attacker receives multiple transfers of rewards, draining the contract's balance.

Recommended Mitigations

- Import and inherit `ReentrancyGuard` from `OpenZeppelin` and add the `nonReentrant` modifier to `claim_many`. Update token_last_balance and time_cursor_of (via _claim) before external calls. Follow the Checks-Effects-Interactions pattern to ensure state changes precede external calls.

Example fix:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract RewardsDistributor is IRewardsDistributor, ReentrancyGuard {
    ...
    function claim_many(uint[] memory _tokenIds) external nonReentrant
    ↪  returns (bool) {
        uint _last_token_time = last_token_time;
        _last_token_time = _last_token_time / WEEK * WEEK;
        address _voting_escrow = voting_escrow;
        uint total = 0;

        for (uint i = 0; i < _tokenIds.length; i++) {
            uint _tokenId = _tokenIds[i];
            if (_tokenId == 0) break;
            uint amount = _claim(_tokenId, _voting_escrow,
↪  _last_token_time); // Updates time_cursor_of
            if (amount != 0) {
                total += amount; // Accumulate total before transfer
                IVotingEscrow.LockedBalance memory _locked =
↪  IVotingEscrow(_voting_escrow).locked(_tokenId);
                if(_locked.end < block.timestamp && !_locked.isPermanent){
                    address _nftOwner =
↪  IVotingEscrow(_voting_escrow).ownerOf(_tokenId);
```

```
                    token_last_balance -= amount; // Update state before
↪  transfer
                    IERC20(token).transfer(_nftOwner, amount); // Transfer
                    ↪  last
                } else {
                    IVotingEscrow(_voting_escrow).deposit_for(_tokenId,
                    ↪  amount);
                    token_last_balance -= amount; // Update state after
↪  deposit
                }
            }
        }
        return true;
    }
    ...
}
```

[M-9] Unchecked External Call to `RewardsDistributor::avm.getOriginalOwner` [Potential Denial of Service]

Description The claim and `claim_many` functions call `avm.getOriginalOwner(_tokenId)` without handling potential reverts when the token's owner is the Auto Voting Manager (AVM). A malicious or faulty avm contract could revert, preventing users from claiming rewards and causing a denial of service.

Impact

Medium Severity: Users with tokens managed by the AVM may be unable to claim rewards, locking funds in the contract and reducing protocol functionality.
User Trust: Persistent failures could discourage participation in the reward system, impacting protocol adoption.

Proof of Concept

```
function claim(uint256 _tokenId) external returns (uint256) {
    ...
    if (amount != 0) {
        IVotingEscrow.LockedBalance memory _locked =
↪  IVotingEscrow(voting_escrow).locked(_tokenId);
        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); // Vulnerable
↪  call
            }
            IERC20(token).transfer(_nftOwner, amount);
        } else {
            ...
        }
        token_last_balance -= amount;
    }
    return amount;
}
```

A malicious avm contract reverts on `getOriginalOwner(_tokenId)`.

The claim transaction fails for tokens managed by the AVM, preventing users from receiving rewards. Users are unable to access their earned tokens, leading to a denial of service.

Recommended Mitigations

- Wrap `avm.getOriginalOwner` in a try-catch block to handle reverts gracefully.
- Implement a fallback mechanism to use `IVotingEscrow(voting_escrow).ownerOf(_tokenId)` if `getOriginalOwner` fails.

Example fix:

```
function claim(uint256 _tokenId) external returns (uint256) {
    ...
    if (amount != 0) {
        IVotingEscrow.LockedBalance memory _locked =
↪  IVotingEscrow(voting_escrow).locked(_tokenId);
        if (_locked.end < block.timestamp && !_locked.isPermanent) {
            address _nftOwner =
↪  IVotingEscrow(voting_escrow).ownerOf(_tokenId);
            if (address(avm) != address(0) && avm.tokenIdToAVMId(_tokenId)
            ↪  != 0) {
                try avm.getOriginalOwner(_tokenId) returns (address
                ↪  originalOwner) {
                    _nftOwner = originalOwner;
                } catch {
                    // Fallback to ve owner if avm call fails
                    emit OriginalOwnerCallFailed(_tokenId);
                }
```

```
            }
            IERC20(token).transfer(_nftOwner, amount);
        } else {
            IVotingEscrow(voting_escrow).deposit_for(_tokenId, amount);
        }
        token_last_balance -= amount;
    }
    return amount;
}
```

Low

[L-1] Missing Zero Address Validation in Critical `AlgebraPoolAPIStorage::initialize` Function

Description

The `AlgebraPoolAPIStorage::initialize` function doesn't validate that the critical addresses (`_algebraFactory` and `_permissionRegistry`) are not zero addresses.

Impact - Complete System Failure: If initialized with zero addresses, all dependent functions will fail

- Permanent Brick: Since this is an upgradeable contract, wrong initialization could permanently disable the contract
- Access Control Bypass: Zero permissionRegistry could cause the CLPoolAdmin modifier to behave unexpectedly

Recomended mitigations Add zero address validation:

```
function initialize(address _algebraFactory, address _permissionRegistry)
↪   public initializer {
    require(_algebraFactory != address(0), "Zero factory");
    require(_permissionRegistry != address(0), "Zero registry");
    __Ownable_init();
    algebraFactory = IAlgebraCLFactory(_algebraFactory);
    permissionRegistry = _permissionRegistry;
}
```

[L-2] AlgebraPoolAPIStorage::CLPoolAdmin‘- External Call Without Validation in Access Control Modifier

Description The `CLPoolAdmin` modifier makes an external call to permissionRegistry without validating that it’s a valid contract:

```
modifier CLPoolAdmin() {
    re-
    ↪  quire(IPermissionsRegistry(permissionRegistry).hasRole("CL_POOL_ADMIN",msg.sende
    ↪  'CL_POOL_ADMIN');
    _;
}
```

Impact - DoS Attack: If `permissionRegistry` points to a contract that reverts, all admin functions become unusable - Access Control Failure: If registry is compromised or returns unexpected values, access control fails

Recomended mitigations

```
modifier CLPoolAdmin() {
    require(permissionRegistry != address(0), "Registry not set");
    require(permissionRegistry.code.length > 0, "Registry not contract");
    re-
    ↪  quire(IPermissionsRegistry(permissionRegistry).hasRole("CL_POOL_ADMIN",msg.sende
    ↪  'CL_POOL_ADMIN');
    _;
}
```

[L-3] `GenesisPoolAPI::getGenesisPoolFromNative` Returns Empty Struct on Zero Address

Description When `genesisPoolFactory.getGenesisPool(nativeToken)` returns `address(0)`, the function returns an empty `GenesisData` struct, which might be confusing for callers expecting either valid data or a clear error.

```
function getGenesisPoolFromNative(address _user, address nativeToken)
↪   external view returns (GenesisData memory genesisData){
    address genesisPool = genesisPoolFactory.getGenesisPool(nativeToken);
    if(genesisPool == address(0)) return genesisData; // Returns empty
    ↪  struct

    return _getGenesisPool(_user, genesisPool);
}
```

Impact

- Ambiguous return values - empty struct vs actual empty pool data
- Difficult for callers to distinguish between “no pool found” and “pool with no data”
- Could lead to integration bugs

Recommeded Mitigations 1. Revert on not found:

```
function getGenesisPoolFromNative(address _user, address nativeToken)
↪   external view returns (GenesisData memory genesisData){
    address genesisPool = genesisPoolFactory.getGenesisPool(nativeToken);
    require(genesisPool != address(0), "No genesis pool found for token");
    return _getGenesisPool(_user, genesisPool);
}
```

2. Return boolean flag:

```
function getGenesisPoolFromNative(address _user, address nativeToken)
   external view returns (bool found, GenesisData memory genesisData){
   address genesisPool = genesisPoolFactory.getGenesisPool(nativeToken);
   if(genesisPool == address(0)) return (false, genesisData);
   return (true, _getGenesisPool(_user, genesisPool));
}
```

Informational

[I-01] Improper Validation in `GenesisPoolAPI::initialize` function

Description The `initialize` function lacks zero-address checks for critical dependencies (_genesisManager, _genesisPoolFactory).

Impact - Contract deployment with invalid dependencies

- Permanent bricking requiring redeployment
- Loss of initialization capability

Recomended mitigations

```
event OwnershipTransferred(address indexed previousOwner, address indexed
↪   newOwner);

function initialize(address _genesisManager, address _genesisPoolFactory)
↪   initializer public {
```

```
        require(_genesisManager != address(0), "GenesisManager cannot be zero
        ↪  address");
        require(_genesisPoolFactory != address(0), "GenesisPoolFactory cannot
        ↪  be zero address");

        address previousOwner = owner;
        owner = msg.sender;

        genesisManager = IGenesisPoolManager(_genesisManager);
        genesisPoolFactory = IGenesisPoolFactory(_genesisPoolFactory);

        emit OwnershipTransferred(previousOwner, msg.sender);
    }
```