SMART CONTRACT AUDIT REPORT

for

BlackholeDEX (Algebra Pools)

Prepared By: Xiaomi Huang

PeckShield

May 24, 2025

Document Properties

Client	BlackholeDEX
Title	Smart Contract Audit Report
Target	BlackholeDEX
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 24, 2025	Xuxian Jiang	Final Release
1.0-rc	April 12, 2025	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the BlackholeDEX protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About BlackholeDEX

BlackholeDEX is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from Solidly/Velodrome with a unique AMM. This audit covers the unique support of adding custom Algebra pools. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of BlackholeDEX

Item	Description
Name	BlackholeDEX
Website	https://blackhole.xyz/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 24, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the given repository will interact with external Algebra pools and this audit does not cover the Algebra pools.

- https://github.com/BlackHoleDEX/SmartContracts.git (52e33af, 0d43a8e)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/BlackHoleDEX/SmartContracts.git (8585039)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `BlackholeDEX` protocol smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	3	■ ■ ■
Low	5	■ ■ ■ ■ ■
Informational	0	
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerability, and 5 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Proposal Cancellation Logic in BlackGovernor	Business Logic	Resolved
PVE-002	Low	Possible Denial-of-Service in Genesis Pool Approval	Business Logic	Resolved
PVE-003	Medium	Revisited _burn() Logic in VotingEscrow	Business Logic	Resolved
PVE-004	High	Incorrect Fee-Claiming Logic in GaugeCL	Business Logic	Resolved
PVE-005	Medium	Lack of _periodFinish Update Upon Reward Notification in GaugeCL	Business Logic	Resolved
PVE-006	Low	Incorrect getReward() Logic in GaugeCL	Business Logic	Resolved
PVE-007	Low	Incorrect ve_for_at() Logic in RewardsDistributor	Business Logic	Resolved
PVE-008	Low	Inconsistent Pair Logic in RouterV2	Business Logic	Resolved
PVE-009	Low	Improved recoverERC20() Logic In GaugeExtraRewarder	Coding Practices	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect Proposal Cancellation Logic in BlackGovernor

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BlackGovernor`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `BlackholeDEX` protocol has the built-in governance support. In the process of reviewing the governance logic in `BlackGovernor`, we notice a business logic issue when canceling a pending proposal.

In the following, we show the implementation of the related `cancel()` routine. It has a rather straightforward logic in validating the given proposal parameters, checking the proposal state, and cancelling the proposal if all conditions for cancellation are met. In particular, there is a need to ensure that only a pending proposal can be cancelled. With that, we need to validate the following statement, i.e., `state(_proposalId)== ProposalState.Pending`, not current `state(proposalId)== ProposalState.Pending` (line 85).

```
67    function cancel(
68        address[] memory targets,
69        uint256[] memory values,
70        bytes[] memory calldatas,
71        bytes32 epochTimeHash
72    ) public virtual override returns (uint256 proposalId) {
73        address proposer = _msgSender();
74        uint256 _proposalId = hashProposal(
75            targets,
76            values,
77            calldatas,
78            epochTimeHash
79        );
80        require(
81            state(proposalId) == ProposalState.Pending,
```

```
            "Governor: too late to cancel"
        );
        require(
            proposer == _proposals[_proposalId].proposer,
            "Governor: only proposer can cancel"
        );
        return _cancel(targets, values, calldatas, epochTimeHash);
    }
```

Listing 3.1: `BlackGovernor::cancel()`

Recommendation Improve the above routine to properly validate a pending proposal for cancellation.

Status This issue has been fixed in the following commit: `abf57b8`.

3.2 Possible Denial-of-Service in Genesis Pool Approval

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GenesisPoolManager`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

`BlackholeDEX` has the unique support of a special pool type, i.e., `genesis pool`. This type of pool is proposed to facilitate new token launch by allowing for crowd-sourcing. While reviewing the lifecycle of `genesis pools`, we notice a possible denial-of-service issue that may block a `genesis pool` from being launched.

In the following, we show the implementation of the related `approveGenesisPool()` routine. This routine is used to approve a `genesis pool` and move the pool to a new `PRE_LISTING` state. The pool approval will create a new pair address (line 151) based on `nativeToken`, `fundingToken`, and `stable` parameters. However, if the new pair creation is not successful, the pool approval transation will be reverted, hence the respective `genesis pool` is blocked.

```
143    function approveGenesisPool(address nativeToken) external Governance {
144        require(nativeToken != address(0), "0x native");
145        address genesisPool = genesisFactory.getGenesisPool(nativeToken);
146        require(genesisPool != address(0), '0x pool');

148        GenesisInfo memory genesisInfo =  IGenesisPool(genesisPool).getGenesisInfo();
149        require(genesisInfo.startTime + genesisInfo.duration - BlackTimeLibrary.
            NO_GENESIS_DEPOSIT_WINDOW > block.timestamp, "time");
```

```
        address pairAddress = pairFactory.createPair(nativeToken, genesisInfo.
            fundingToken, genesisInfo.stable);
        pairFactory.setGenesisStatus(pairAddress, true);

        liveNativeTokens.push(nativeToken);
        liveNativeTokensIndex[nativeToken] = liveNativeTokens.length; // because default
             value is 0, so starting with 1

        IGenesisPool(genesisPool).approvePool(pairAddress);
    }
```

Listing 3.2: `GenesisPoolManager::approveGenesisPool()`

Recommendation Improve the above routine to explicitly check the pair presence and only create a new one if it does not exist.

Status This issue has been fixed in the following commit: `e29ad71`.

3.3 Revisited _burn() Logic in VotingEscrow

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `VotingEscrow`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `BlackholeDEX` protocol has a `veNFT` implementation that escrows `ERC-20 tokens` in the form of an `ERC-721 NFT`. While reviewing the escrow logic, we notice the implementation has an issue when burning an `ERC-721 NFT`.

In the following, we show the implementation of the related `_burn()` routine. It has the most basic functionality in clearing the approval state, adjusting the associated delegation, and then removing the voting token id. However, the delegation adjustment needs to be performed after the token removal, not before. In other words, the call to `moveTokenDelegates()` (line 548) should occur after the `_removeTokenFrom()` call (line 551).

```
540    function _burn(uint _tokenId) internal {
541        require(_isApprovedOrOwner(msg.sender, _tokenId), "IA");

543        address owner = ownerOf(_tokenId);

545        // Clear approval
546        approve(address(0), _tokenId);
```

```
        // checkpoint for gov
        VotingDelegationLib.moveTokenDelegates(cpData, delegates(owner), address(0),
            _tokenId, ownerOf);
        // Remove token
        //_removeTokenFrom(msg.sender, _tokenId);
        _removeTokenFrom(owner, _tokenId);

        emit Transfer(owner, address(0), _tokenId);
    }
```

Listing 3.3: `VotingEscrow::_burn()`

Recommendation Improve the above routine to properly adjust the internal order when a voting token is burned.

Status This issue has been fixed in the following commit: `7a074ff`.

3.4 Incorrect Fee-Claiming Logic in GaugeCL

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `GaugeCL`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `BlackholeDEX` protocol has the unique support of `Algebra` pools. In the process of reviewing the associated gauge logic (in `GaugeCL`) paired with supported `Algebra` pools, we notice a business logic issue when claiming a pending fee.

In the following, we show the implementation of the related `_claimFees()` routine. It has a rather straightforward logic in claiming the pool fee and sending to the gauge-specific internal bribe contract. However, it comes to our attention that the `token1`-associated fee should be calculated as `claimed1 -= _dibsFeeToken1;`, not current `claimed0 -= _dibsFeeToken1;` (line 221).

```
195     function _claimFees() internal returns (uint256 claimed0, uint256 claimed1) {
196         if (!isForPair) {
197             return (0, 0);
198         }

200         address _token0 = algebraPool.token0();
201         address _token1 = algebraPool.token1();
202         // Fetch fee from the whole epoch which just ended and transfer it to internal
                Bribe address.
203         claimed0 = IERC20(_token0).balanceOf(address(this));
```

```
        claimed1 = IERC20(_token1).balanceOf(address(this));

        if (claimed0 > 0  claimed1 > 0) {
            // Deduct dibsPercentage from fee accrued and transfer to dibs address(
                Foundation address)

            uint256 referralFee = IGaugeFactoryCL(factory).dibsPercentage();
            address dibs = IGaugeFactoryCL(factory).dibs();
            uint256 _dibsFeeToken0 = (dibs != address(0)) ? (claimed0 * referralFee /
                10000) : 0;
            uint256 _dibsFeeToken1 = (dibs != address(0)) ? (claimed1 * referralFee /
                10000) : 0;

            if (_dibsFeeToken0 > 0) {
                _safeTransfer(_token0, dibs, _dibsFeeToken0); // Transfer dibs fees
                claimed0 -= _dibsFeeToken0;
            }

            if (_dibsFeeToken1 > 0) {
                _safeTransfer(_token1, dibs, _dibsFeeToken1); // Transfer dibs fees
                claimed0 -= _dibsFeeToken1;
            }

            uint256 _fees0 = claimed0;
            uint256 _fees1 = claimed1;
            ...
        }
        ...
    }
```

Listing 3.4: `GaugeCL::_claimFees()`

Recommendation Improve the above routine to properly claim the pool fees.

Status This issue has been fixed in the following PR: `180`.

3.5 Lack of _periodFinish Update Upon Reward Notification in GaugeCL

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GaugeCL`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned earlier, `BlackholeDEX` has the unique support of `Algebra` pools, which greatly expands the liquidity reach to more recent `DEX` engines. While reviewing the logic to add extra rewards, we notice an issue that does not properly update the reward parameter, `_periodFinish`, hence corrupting subsequent reward dessimination.

In the following, we show the implementation of the related `notifyRewardAmount()` routine. This routine is used to transfer emission to farming virtual pool address. While current implementation has properly updated the reward rates and synchronized with the associated `Algebra virtual pool`, it does not update another important risk parameter, `_periodFinish`. This lack of udpate may completely mess up the subsequent reward dissemination.

```
    function notifyRewardAmount(address token, uint256 reward) external nonReentrant
        isNotEmergency onlyDistribution {
        require(token == address(rewardToken), "not rew token");
        // Transfer emission to Farming Virtual Pool address
        if (block.timestamp >= _periodFinish) {
            rewardRate = reward / DURATION;
        } else {
            uint256 remaining = _periodFinish - block.timestamp;
            uint256 leftover = remaining * rewardRate;
            rewardRate = (reward + leftover) / DURATION;
        }
        (IERC20Minimal rewardTokenAdd, IERC20Minimal bonusRewardTokenAdd, IAlgebraPool
            pool, uint256 nonce) =
                algebraEternalFarming.incentiveKeys(poolAddress);
        IncentiveKey memory incentivekey = IncentiveKey(rewardTokenAdd,
            bonusRewardTokenAdd, pool, nonce);
        bytes32 incentiveId = IncentiveId.compute(incentivekey);

        // set RewardRate to AlgebraVirtual Pool
        (,,address virtualPoolAddress,,,) = algebraEternalFarming.incentives(incentiveId
            );
        (,uint128 bonusRewardRate) = IAlgebraEternalVirtualPool(virtualPoolAddress).
            rewardRates();
```

```
        algebraEternalFarming.setRates(incentivekey, uint128(rewardRate),
            bonusRewardRate);

        // transfer emission Reward to Algebra Virtual Pool
        algebraEternalFarming.addRewards(incentivekey, uint128(reward), 0);
        emit RewardAdded(reward);
    }
```

Listing 3.5: `GaugeCL::notifyRewardAmount()`

Recommendation Improve the above routine to timely update all related reward parameters, including `_periodFinish`.

Status This issue has been fixed in the following PR: `180`.

3.6 Incorrect getReward() Logic in GaugeCL

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GaugeCL`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `Algebra pool` support in `BlackholeDEX` has another issue. In particular, the related farming logic allows for the distribution of two reward tokens and one of them is named as bonus reward. When these two reward tokens are being claimed, our analysis shows the logic should be improved.

In the following, we show the implementation of the related `getReward()` routine. The last parameter of this routine, i.e., `isBonusReward`, is used to indicate which reward token is claimed. It comes to our attention that when `isBonusReward = true`, the claim is intended for the bonus reward, not the base one (line 140).

```
137    function getReward(uint256 tokenId, uint256 amountRequested, bool isBonusReward)
           public nonReentrant onlyDistribution {
138        address owner = nonfungiblePositionManager.ownerOf(tokenId);
139        (IERC20Minimal rewardTokenAdd, IERC20Minimal bonusRewardTokenAdd,,) =
               algebraEternalFarming.incentiveKeys(poolAddress);
140        farmingCenter.claimReward(isBonusReward == true ? rewardTokenAdd :
               bonusRewardTokenAdd , owner, amountRequested);
141        emit Harvest(owner, amountRequested);
142    }
```

Listing 3.6: `GaugeCL::getReward()`

Recommendation Improve the above routine to properly claim the intended rewards.

Status This issue has been fixed in the following PR: `180`.

3.7 Incorrect ve_for_at() Logic in RewardsDistributor

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardsDistributor`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

To compensate the users who lock their governance tokens, `BlackholeDEX` has a `RewardsDistributor` contract to provide necessary token emissions. Within this `RewardsDistributor` contract, there is a getter function `ve_for_at()` to query the voting balance of the given `tokenId` at a specific timestamp. Our analysis shows its implementation is inaccurate.

```
    function ve_for_at(uint _tokenId, uint _timestamp) external view returns (uint) {
        address ve = voting_escrow;
        uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
        uint epoch = _find_timestamp_user_epoch(ve, _tokenId, _timestamp, max_user_epoch
            );
        IVotingEscrow.Point memory pt = IVotingEscrow(ve).user_point_history(_tokenId,
            epoch);
        return Math.max(uint(int256(pt.bias - pt.slope * (int128(int256(_timestamp - pt.
            ts)))) + int256(pt.permanent + pt.smNFT + pt.smNFTBonus)), 0);
    }
```

Listing 3.7: `RewardsDistributor::ve_for_at()`

To elaborate, we show above the implementation of this `ve_for_at()` routine. While it properly retrieves the user balance in `IVotingEscrow.Point`, the voting power needs to be properly computed. In particular, we need to differentiate the lock type, i.e., `smNFT`, `permanent`, or `decaying`. An example revision is shown as below:

```
    function ve_for_at(uint _tokenId, uint _timestamp) external view returns (uint) {
        address ve = voting_escrow;
        uint max_user_epoch = IVotingEscrow(ve).user_point_epoch(_tokenId);
        uint epoch = _find_timestamp_user_epoch(ve, _tokenId, _timestamp, max_user_epoch
            );
        IVotingEscrow.Point memory pt = IVotingEscrow(ve).user_point_history(_tokenId,
            epoch);


        if (pt.smNFT != 0){
```

```
            return pt.smNFT + pt.smNFTBonus;
        }
        else if (pt.permanent != 0) {
            return lpt.permanent;
        }
        else {
            pt.bias -= pt.slope * int128(int256(_timestamp) - int256(pt.ts));
            if (pt.bias < 0) {
                lpt = 0;
            }
            return uint(int256(pt.bias));
        }
    }
```

Listing 3.8: Revised `RewardsDistributor::ve_for_at()`

Recommendation Improve the above routine to properly calculate the voting balance of a given `tokenId` at a specific timestamp.

Status This issue has been resolved as the above function has been removed.

3.8 Inconsistent Pair Logic in RouterV2

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RouterV2`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

To facilitate the token swaps, `BlackholeDEX` protocol has a convenient helper contract, i.e., `RouterV2`. This router contract has properly supported the `stable/volatile` pairs. However, the new support of `Algebra` pools with concentrated liquidity requires necessary revision to the `RouterV2` contract.

In the following, we show the implementation of an example `swapExactTokensForTokensSimple()` routine. As the name indicates, this routine is used to swap a given input token for the intended output token. And current implementation only supports the `stable/volatile` pair, but not concentrated pools (line 557). Note it also affects other routines, including `swapExactETHForTokens()`, `swapExactTokensForETH()`, and `_swapSupportingFeeOnTransferTokens()`.

```
541     function swapExactTokensForTokensSimple(
542         uint amountIn,
543         uint amountOutMin,
544         address tokenFrom,
545         address tokenTo,
```

```
        bool stable,
        address to,
        uint deadline
    ) external ensure(deadline) returns (uint[] memory amounts) {
        route[] memory routes = new route[](1);
        routes[0].from = tokenFrom;
        routes[0].to = tokenTo;
        routes[0].stable = stable;
        amounts = getAmountsOut(amountIn, routes);
        require(amounts[amounts.length - 1] >= amountOutMin, 'BaseV1Router:
            INSUFFICIENT_OUTPUT_AMOUNT');
        _safeTransferFrom(
            routes[0].from, msg.sender, pairFor(routes[0].from, routes[0].to, routes[0].
                stable), amounts[0]
        );
        _swap(amounts, routes, to);
    }
```

Listing 3.9: `RouterV2::swapExactTokensForTokensSimple()`

Recommendation Improve the above routines to properly support concentrated pools.

Status This issue has been resolved in the following commit: `37ad1e2`.

3.9 Improved recoverERC20() Logic In GaugeExtraRewarder

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GaugeExtraRewarder`
- Category: Coding Practices [3]
- CWE subcategory: CWE-663 [1]

Description

The `gauge` support in `BlackholeDEX` protocol allows for the use of extra rewards. In the process of examining the extra rewarding mechanism, we notice an issue when recovering the funds from the contract.

In the following, we show the implementation of the related `recoverERC20()` routine. It has a rather straightforward logic in recovering the funds in the contract. However, when the token to be recovered is the intended reward token, it has an implicit assumption that `lastDistributedTime` is not less than current timestamp, i.e., `block.timestamp`. Otherwise, the computation of time left will be reverted (line 183). This assumption is not necessary and should be eliminated.

```
174    function recoverERC20(uint amount, address token) external onlyOwner {
175        require(amount > 0, "amount > 0");
```

```
        require(token != address(0), "addr0");
        uint balance = IERC20(token).balanceOf(address(this));
        require(balance >= amount, "not enough tokens");

        // if token is = reward and there are some (rps > 0), allow withdraw only for remaining rewards and then set new rewPerSec
        if(token == address(rewardToken) && rewardPerSecond != 0){
            updatePool();
            uint timeleft = lastDistributedTime - block.timestamp;
            uint notDistributed = rewardPerSecond * timeleft;
            require(amount <= notDistributed, 'too many rewardToken');
            rewardPerSecond = (notDistributed - amount) / timeleft;
        }
        IERC20(token).safeTransfer(msg.sender, amount);

    }
```

Listing 3.10: `GaugeExtraRewarder::recoverERC20()`

Recommendation Improve the above routine to properly remove unwanted assumption.

Status This issue has been resolved in the following commit: `8585039`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `BlackholeDEX` protocol, which is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from `Solidly/Velodrome` with a unique `AMM`. This audit covers the unique support of adding custom `Algebra` pools. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[3] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[5] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.