Security Review

Cantina Managed

TRANTOR: GIGA-DEX

Cantina Managed review by:
Valerian Callens, Lead Security Researcher

CANTINA

September 3, 2026

Contents

1 Introduction

1.1 About Cantina

Cantina is a security platform that pairs a world-class network of security researchers with agentic AI to help teams building onchain find and fix real risk. Through managed reviews, competitions, and bounties, Cantina secures protocols before vulnerabilities can be exploited.
Learn more at cantina.security

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Aug 15th to Aug 22nd the Cantina team conducted a review of apex-contracts on commit hash 6d36108c. The team identified a total of **6** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	4	0	4
Gas Optimizations	0	0	0
Informational	2	0	2
Total	**6**	**0**	**6**

2.1 Scope

The security review had the following components in scope for apex-contracts on commit hash 6d36108c:

```
src/core
├── ApexController.sol
├── ApexToken.sol
├── ApexVault.sol
├── EmissionCenter.sol
├── FeeCenter.sol
├── FeeReceiver.sol
├── libraries/ApexControllerFeePolicy.sol
└── storage
    ├── ApexControllerStorage.sol
    ├── ApexVaultStorage.sol
    ├── FeeCenterStorage.sol
    ├── FeeReceiverStorage.sol
    └── VeApexTokenStorage.sol
```

In addition, the Cantina team performed a diff review between:

- `src/core/ClassicChef.sol` and https://github.com/lfj-gg/joe-core/blob/main/contracts/MasterChefJoeV2.sol at commit 5bec2f3;
- `src/core/CLMasterChef.sol` and https://github.com/pancakeswap/pancake-v3-contracts/blob/main/projects/masterchef-v3/contracts/MasterChefV3.sol at commit 9868479;
- `src/core/LMPool.sol` and https://github.com/pancakeswap/pancake-v3-contracts/blob/main/projects/v3-lm-pool/contracts/PancakeV3LmPool.sol at commit 9868479;
- `src/periphery/NonfungiblePositionManager.sol` and https://github.com/pancakeswap/pancake-v3-contracts/blob/main/projects/v3-periphery/contracts/NonfungiblePositionManager.sol at commit 9868479;

2.2 Codebase context

Apex is a Classic and concentrated-liquidity (CL) DEX. It combines Solidly-style ERC20 pools with a Pancake-compatible CL stack and Apex-specific controller, emission, fee-routing, veNFT, and vault systems.

Current Architecture:

- `ClassicPool` / `ClassicFactory`: volatile and stable ERC20 LP pools.
- `ClassicRouter`: Classic liquidity operations and atomic add-and-stake flows.
- `CLPool` / `CLFactory` / `CLPoolDeployer`: concentrated-liquidity pools and deployment.
- `NonfungiblePositionManager`, routers, quoters and lens contracts.
- `ClassicChef` / `CLMasterChef`: controller-funded rewards for Classic LPs and CL NFTs.
- `ApexController`: upgradeable protocol authority for setup, fees, farms, emissions, treasury routing, and contract administration.
- `EmissionCenter`: vesting source for APEX emissions released to the controller.
- `FeeCenter` / `FeeReceiver`: protocol-fee intake, receiver registry, fee splitting, approved swaps, and Vault notifications.
- `VeApexToken` / `ApexVault`: upgradeable lock NFT and multi-token reward streaming/compound system.

Source Structure:

- Classic contracts follow Solidly and JoeV2/Sushi-style pool and MasterChef patterns.
- CL core, callbacks, periphery, and LM contracts follow Pancake V3 patterns, with Apex controller-driven emissions and MEV-resistant farming.
- Apex-owned upgradeable state is stored in namespaced libraries under `src/core/storage`.

Authority and Upgrades:

- `ApexController` is the runtime role hub.
- `DEFAULT_ADMIN_ROLE` controls protocol configuration, farms, treasury, receivers, rescue paths, and role management.
- `FEE_OPERATOR` manages fee overrides, fee collection, splitting, and receiver operations.
- `EMISSION_OPERATOR` manages emission windows.
- `VAULT_OPERATOR` manages Vault notifications and compound operations.

`ApexController`, `VeApexToken`, `ApexVault`, and `FeeCenter` use OpenZeppelin transparent proxies. Proxy administration is separate from controller roles.

2.3 Trust assumptions

The review was conducted under the following trust assumptions:

Roles and permissions:

- Privileged actors (`DEFAULT_ADMIN_ROLE`, `FEE_OPERATOR`, `EMISSION_OPERATOR`, `VAULT_OPERATOR`) are assumed to perform their duties in accordance with documented procedures, mandates, and responsibilities. These actors are assumed to perform appropriate due diligence before authorizing on-chain actions.
- The review assumes adherence to the principle of Segregation of Duties. No single actor is expected to simultaneously hold operational roles intended to remain independent, and collusion between incompatible roles is considered out of scope unless explicitly stated otherwise.
- End users are considered untrusted and may behave arbitrarily or maliciously within the permissions granted by the system. Users are responsible for reviewing and understanding the on-chain actions they submit.

Acknowledged self-identified issues shared with the Cantina team:

- F-01 (Info): Pancake-style CL emergency mode can skip LM cleanup; emergency withdrawal is intentionally minimal and reward-agnostic.
- F-02 (Low): Pancake-style CL collect/sweep behavior is retained; `collect` and `collectTo` can sweep full Chef-held balances. Raw callers are responsible for cleanup and safe token usage.
- F-03 (Low): Raw NFP `mint` remains possible and can bypass active-farm staking UX, so staked-farm users should use `mintAndStake`.
- F-04 (Low): SmartRouter direct swaps do not enforce per-function deadlines by themselves; deadline/validation is enforced through recommended multicall patterns.
- F-05 (Info): Classic TWAP views can revert without sufficient history; integrations should check availability and fallback.
- F-06 (Info): Classic protocol-fee routing for mint-settled lazy fees uses current pair config at settlement time.
- F-07 (Info): `rebalanceEmissions` updates both Chef books together; gas is bounded by current farm-count assumptions.
- F-08 (Info): Deployment is non-atomic across broadcasted txs; validated scripts must be used before launch usage.
- F-09 (Info): VeApex token URI/view behavior follows veAERO non-strict ERC721 semantics.
- F-10 (Low): CL `pendingReward` is checkpoint-state based (not a live-forwarded projection).
- F-11 (Low): Vault reward notify/reveal functions require non-zero active bucket weight.
- F-12 (Low): Removed FeeCenter receivers remain known for leftover cleanup and operator-managed rescue paths.
- F-14 (Low): Vault inactivity is enforced lazily through `flushInactive`, so expired positions remain active until flushed.
- F-15 (Medium): `FEE_OPERATOR` can change fee economics within protocol limits; trusted-role assumption applies.
- F-16 (Low): Vault per-user settlement can leave bounded dust due to stream rounding.
- F-17 (Info): VeApex global checkpoint lag can persist past 255 weeks if unchecked; accepted as ve-style edge-cadence.
- F-19 (Low): Classic pool direct quote edge cases from direct `Pair.getAmountOut` behavior remain inherited.
- F-22 (Low): `ClassicChef.emergencyWithdraw` is emergency-only and bypasses reward checkpointing by design.
- F-23 (Low): Direct veNFT safe transfer to Vault without `stake` is user-integration error and unsupported.
- F-24 (Low): Public periphery cleanup (`refundETH`, `unwrapWETH9`, `sweepToken`) is inherited and relies on trusted token behavior and safe caller pattern.
- F-26 (Low): FeeCenter splitting scales linearly with active receivers; governance should keep receiver sets bounded.
- F-27 (Low): Active farm status does not guarantee non-zero emission allocation.
- F-28 (Low): Classic pair creation assumes standard ERC-20 metadata and sane decimals.
- F-29 (Info): Bytecode size is constrained; late additions must stay within deployment size checks.
- F-30 (Low): Some fee-receiver notify helpers revert on zero balances by design.
- F-31 (Low): Direct `VeApexToken.depositFor` on Vault-staked IDs can stale Vault weights until Vault interaction refreshes.
- F-33 (Info): Some deploy/wiring checks are canonical-script enforced rather than per-contract repeated checks.

- F-35 (Low): Deactivating a farm resets pair-farm fee/default routing; prior overrides must be re-applied to continue.
- F-36 (Low): Classic global setters are factory defaults only for new pairs and do not retro-propagate.
- F-37 (Low): Some low-frequency admin surfaces are intended to remain upgrade-only due to bytecode constraints.
- F-38 (Info): `createLockFor` can mint veNFTs directly to non-ERC721-capable recipients; caller is responsible for recipient compatibility.
- F-39 (Info): Classic stable `sync()` retains inherited reserve-domain edge cases; treated as AMM compatibility.
- F-40 (Info): Controller reward-liability can retain tiny bounded raw APEX dust by conservative accounting.
- F-41 (Medium): Aggregator swap execution assumes documented allowance spender and calldata target alignment.
- F-42 (Low): Vault unstakes settle all reward tokens; a reverted reward token can block unstake and relies on governed token trust model.

Integration recommendations:

- Use SDK call-parameter helpers (`ApexClassicRouter`, `ApexSmartRouter`, `ApexCLFarm`) instead of raw ABI calls for normal user flows.
- For CL farm deposits, use `mintAndStake` and pass the end-user as staker/beneficiary.
- Prefer atomic multicall paths with deadline/previous-blockhash safeguards for swaps.
- Use Chef `collectTo` helpers and avoid exposing raw NFP/CL collect/sweep flows as default UX.
- Perform expected cleanup (`refundETH`, `unwrapWETH9`, `SWEEP_TOKEN`) in expected order for periphery operations that can leave residual balances.
- Validate lock and pool participants with vault-aware flows (`stake`, `increaseAmount`) instead of direct raw `depositFor`/raw vault transfers.

2.4 Cantina Managed Team Statement

Trantor engaged Cantina to conduct a security review of its Apex contracts. We would like to thank the Trantor team for their responsiveness and constructive engagement throughout the review process.

No significant security issues were identified within the scope of the assessment. During the engagement, the Trantor team demonstrated strong security awareness and proactively identified and addressed numerous edge cases. Their responsiveness and proactive approach to risk management contributed positively to the overall security posture of the reviewed system.

3 Findings

3.1 Low Risk

3.1.1 Classic protocol fee collection can misattribute LP-token fees when a pair token is used as another pair's asset

Severity: Low Risk.

Context: *(No context files were provided by the reviewer).*

Description: `collectClassicProtocolFees(pair)` is intended to realize only that `pair`'s pending protocol-fee LP. However, `FeeCenter.realizeClassicFees(pair)` currently burns the entire `FeeCenter` balance of `pair` LP token.

Classic pair addresses are ERC-20 LP tokens, and `ClassicFactory.createPair()` is permissionless. So a pair `Q` can be created with an existing pair token `P` as `token0` / `token1`. If `Q` protocol fees are collected first, `FeeCenter` can hold raw `P` from `Q`. When `collectClassicProtocolFees(P)` is later called, all `P` held by `FeeCenter` is burned as `P` fees, which can consume those `Q`-origin tokens and convert them into `P`'s underlying assets.

As impact, this is not a direct loss of value, since LP tokens have redeemable value, but it breaks fee-domain integrity and operator expectations:

- Fees can move unexpectedly from LP-domain (`P`) to underlying-asset domain (`A/B`) and lose their original identity.
- `pushAndSplitFees([P, ...])` may omit the intended `P` amount, causing routing mismatch.
- Receiver/vault/reporting flows that assume `P`-denominated fees become incorrect or harder to reconcile.
- In some states, mixed/incorrect burns can cause avoidable settle/reconcile liveness issues (e.g., unexpected revert behavior).

The root cause is that `FeeCenter.realizeClassicFees(pair)` uses `FeeCenter`'s full current `pair`-token balance, not the amount minted by the specific `collectClassicProtocolFees(pair)` call, so unrelated prior `pair` token balances are taken as well.

The following is needed for that scenario to happen:

1. Pair `P` exists.
2. Another pair `Q` is permissionlessly created with `P` as an underlying token.
3. Operator (`FEE_OPERATOR`) collects protocol fees for `Q`, leaving raw `P` in `FeeCenter`.
4. Operator later collects protocol fees for `P`.

Recommendation: Consider implementing delta-based realization in `collectClassicProtocolFees()`:

1. Read `FeeCenter`'s `pair` balance before `mintFee()`.
2. Call `IClassicPool(pair).mintFee()`.
3. Read the balance after and compute `delta`.
4. Realize only `delta` LP in `FeeCenter` (with bounds check).
5. Keep any pre-existing `pair` balance untouched.

Optional hardening: disallow using an existing pair token as a pair input in `createPair()` if this behavior is undesired.

Trantor: Ok, I agree on the technicalities, although operator always collect both at same time atomically, but will add on our next upgrade.

Cantina Managed: Acknowledged.

3.1.2 Mismatch between code and specs for the initial protocol fee of ungauged CL pools

Severity: Low Risk.

Context: *(No context files were provided by the reviewer)*.

Description: The specs state that ungauged pools should start with a protocol fee of 20%:

```
- Apex extends CL protocol-fee range to the full safe `0..10000` scale. Ungauged
↪  pools start at 20%; `DEFAULT_ADMIN_ROLE` can change the global ungauged policy
↪  and `FEE_OPERATOR` can explicitly apply it to selected inactive pools. Activated
↪  farms use 100%.
```

It is the case for classic pools, as it can be seen in `ClassicFactory`:

```
uint256 public constant FEE_DENOM = 1_000_000;
uint256 public constant DEFAULT_PROTOCOL_FEE = FEE_DENOM / 5;
```

However, CL pools are initialized with a protocol fee of 10%:

```
uint32 internal constant DEFAULT_PROTOCOL_FEE = 1_000; // 10% initialization
↪  fallback
uint32 internal constant DEFAULT_PROTOCOL_FEE_PACKED = DEFAULT_PROTOCOL_FEE +
↪  (DEFAULT_PROTOCOL_FEE << 16);

//[...]

    function initialize(uint160 sqrtPriceX96) external override {
        require(slot0.sqrtPriceX96 == 0, "AI");

        int24 tick = TickMath.getTickAtSqrtRatio(sqrtPriceX96);

        (uint16 cardinality, uint16 cardinalityNext) =
        ↪  observations.initialize(_blockTimestamp());

        slot0 = Slot0({
            sqrtPriceX96: sqrtPriceX96,
            tick: tick,
            observationIndex: 0,
            observationCardinality: cardinality,
            observationCardinalityNext: cardinalityNext,
            feeProtocol: DEFAULT_PROTOCOL_FEE_PACKED,
            unlocked: true
        });

        emit Initialize(sqrtPriceX96, tick);
    }
```

Recommendation: Consider clarifying if CL pools should be initialized with a protocol fee of 10% or 20%, and adapt the code or the specs accordingly.

Trantor: True. All should default to 10%.

Cantina Managed: Acknowledged.

3.1.3 `collectClassicProtocolFees()` silently returns zero fees when fee receiver is overridden

Severity: Low Risk.

Context: ApexControllerFeePolicy.sol#L83-L88.

Description: Classic fee collection assumes `s.feeCenter` is always the fee receiver of `pair`, but the fee receiver can be replaced via `overrideClassicFeeReceiver()`.

`collectClassicProtocolFees(pair)` calls `mintFee()` and then invokes `FeeCenter.realizeClassicFees(pair)`. However, `ClassicPool.mintFee()` sends newly minted protocol-fee LP tokens to the pair's configured `feeReceiver`.

As a result, if `overrideClassicFeeReceiver(pair, alternateReceiver)` has been used, those LP tokens are sent to `alternateReceiver`, while `s.feeCenter` has no corresponding balance. `realizeClassicFees()` then returns `(0, 0)` without reverting, so the collection call silently reports no fees and the newly minted fees remain with the `alternateReceiver`:

```
function realizeClassicFees(address _pair)
    external
    onlyApexController
    nonReentrant
    returns (uint256 amount0, uint256 amount1)
{
    require(_pair != address(0), ApexErrors.ZERO_ADDRESS());
    uint256 liquidity = IERC20(_pair).balanceOf(address(this));
    if (liquidity == 0) return (0, 0);

    IERC20(_pair).safeTransfer(_pair, liquidity);
    (amount0, amount1) = IClassicPool(_pair).burn(address(this));

    emit ClassicFeesRealized(_pair, liquidity, amount0, amount1);
}
```

Recommendation: Consider treating separately in `collectClassicProtocolFees()` pairs where `feeCenter` is not the fee receiver.

Trantor: I don't think it matters, fee operator relay just waits for balances to go up before distributing.

Cantina Managed: Acknowledged.

3.1.4 Exact-expiry compound can return inflated amount and reweight an expired position

Severity: Low Risk.

Context: *(No context files were provided by the reviewer)*.

Description: In `ApexVault._compoundPosition()`, the function reads `amount = s.rewards[_tokenId][address(s.veNFT)]` before calling `_compoundStored(_tokenId)`, but it does not use `_compoundStored()`'s return value:

```
function _compoundPosition(uint256 _tokenId) internal returns (uint256 amount) {
        ApexVaultStorage.Layout storage s = ApexVaultStorage.layout();
        ApexVaultStorage.Position memory position = s.positions[_tokenId];
        if (!position.staked || !position.active || !_isEligible(_tokenId)) return
        ↪  0;

        _settleAllRewards(_tokenId);
        amount = s.rewards[_tokenId][address(s.veNFT)];
```

```
        if (amount == 0) return 0;

        _removeWeights(_tokenId);
        _compoundStored(_tokenId);
        _poke(_tokenId);
        s.positions[_tokenId].active = true;
        s.activePositions.add(_tokenId);
    }
```

It becomes problematic in the specific edge case when `minRemainingLock == 0` and we are at the exact expiry (`lock.end == block.timestamp`): `_isEligible()` returns `true` , so `_compoundPosition()` proceeds, while `_compoundStored()` treats the position as expired (`lock.end <= block.timestamp`) and forfeits to treasury, returning `0` to `_compoundPosition()` :

```
function _isEligible(uint256 _tokenId) internal view returns (bool) {
    IVotingEscrow veNFT_ = ApexVaultStorage.layout().veNFT;
    try veNFT_.locked(_tokenId) returns (IVotingEscrow.LockedBalance memory lock) {
        return !lock.isPermanent && lock.amount > 0
            && lock.end >= block.timestamp +
            ↪  ApexVaultStorage.layout().minRemainingLock;
    } catch {
        return false;
    }
}

function _compoundStored(uint256 _tokenId) internal returns (uint256 amount) {
    ApexVaultStorage.Layout storage s = ApexVaultStorage.layout();
    address compoundBucket = address(s.veNFT);
    amount = s.rewards[_tokenId][compoundBucket];
    if (amount == 0) return 0;

    s.rewards[_tokenId][compoundBucket] = 0;
    IVotingEscrow.LockedBalance memory lock = s.veNFT.locked(_tokenId);
    if (lock.amount <= 0 || lock.end <= block.timestamp) {
        IERC20(s.veNFT.token()).safeTransfer(IApexController(s.apexController).trea
        ↪  sury(),
        ↪  amount);
        emit CompoundForfeited(s.positions[_tokenId].owner, _tokenId, amount);
        return 0;
    }
```

In that case:

- The stale pre-read `amount` can still be returned by `compound()` / `compoundMany()` ;
- `_poke()` is still executed, reweighting an effectively forfeited position.

That edge case remains unlikely, since setting `minRemainingLock` to `0` is a privileged action and would discard that behavior.

Recommendation: Consider the following:

1. Return and validate the value returned by `_compoundStored()` in `_compoundPosition()` ;
2. Only execute `_poke()` when compounding was actually processed;
3. Unify expiry semantics between `_isEligible()` and `_compoundStored()` so the same boundary is used consistently.

Trantor: `minRemainingLock == 0` will be impossible while still earning reward because people will get the flush inactive after <5 months to stop earning triggered by the relay bots, so I think this doesn't matter. Will still add just in case though on our next upgrade.

Cantina Managed: Acknowledged.

3.2 Informational

3.2.1 `stopAllEmissions()` does not emit an event, reducing system observability

Severity: Informational.

Context: *(No context files were provided by the reviewer).*

Description: The `stopAllEmissions()` function in `ApexController` is a privileged action restricted to the `EMISSION_OPERATOR` role that force-stops emissions:

```
function stopAllEmissions() external onlyRole(EMISSION_OPERATOR) {
    ApexControllerStorage.Layout storage $ = ApexControllerStorage.layout();
    $.classicChef.massUpdatePools();
    $.clMasterChef.massUpdatePools();

    if ($.latestPeriodEndTime > block.timestamp) {
        $.rewardLiability -= (($.latestPeriodEndTime - block.timestamp) *
        ↪  $.latestPeriodRewardPerSecond)
            / REWARD_PRECISION;
    }
    $.latestPeriodEndTime = block.timestamp;
    $.latestPeriodRewardPerSecond = 0;
}
```

However, the function does not emit an event indicating that emissions were forcibly stopped, the address that initiated the action, or the resulting emission state.

As a result, administrators, indexers, monitoring systems, and incident-response tooling cannot identify an emergency emission halt directly from the event stream. This reduces the observability and auditability of privileged emission-control actions.

Recommendation: Consider emitting a dedicated event when `stopAllEmissions()` is called, including the caller, and any relevant pre and post state.

Trantor: Agree but we won't track it historically so no need.

Cantina Managed: Acknowledged.

3.2.2 `_claimRewards()` accepts non-configured token addresses, leading to unnecessary state and gas usage

Severity: Informational.

Context: ApexVault.sol#L474-L481.

Description: `_claimRewards()` in `ApexVault.sol` only blocks the addresses `address(veNFT)` and `veNFT.token()`, but accepts any other token and further calls `_settleReward(tokenId, token)` then `_payReward()`.

For a position that did not configure that token, this results in no payout (as expected), but it still writes `userRewardPerWeightPaid[tokenId][token]` and may touch unrelated reward buckets with no user benefit, resulting in state-noise and unnecessary gas consumption.

Recommendation: Consider adding the following check in the `_claimRewards()` loop:

```
require(s.configs[_tokenId][_tokens[i]].bps != 0, ApexErrors.INVALID_ASSORTMENT());
```

Trantor: Agree.

Cantina Managed: Acknowledged.