SALUS SECURITY

JAN 2026

CODE SECURITY ASSESSMENT

LORENZO

Overview

Project Summary

- Name: Lorenzo - Earn Manager
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
    - https://github.com/Lorenzo-Protocol/earn-contract
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Lorenzo - Earn Manager
Version	v2
Type	Solidity
Dates	Jan 19 2026
Logs	Jan 15 2026, Jan 19 2026

Vulnerability Summary

Total High-Severity issues	3
Total Medium-Severity issues	2
Total Low-Severity issues	3
Total informational issues	1
Total	9

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 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 with 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.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Potential Dos in rebalance via Direct Liquidity Donation	High	Business Logic	Resolved
2	Ineffective Slippage Protection in rebalanceLiquidity	High	Business Logic	Resolved
3	Users may fail to add liquidity	High	Business Logic	Resolved
4	Stale pcsPool Reference Following Pool Fee Configuration Update	Medium	Business Logic	Resolved
5	Centralization risk	Medium	Business Logic	Acknowledged
6	Deadline check does not work	Low	Business Logic	Acknowledged
7	Missing input validation in mint	Low	Business Logic	Resolved
8	Incorrect current price calculation	Low	Business Logic	Resolved
9	Redundant code	Informational	Configuration	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Potential Dos in rebalance via Direct Liquidity Donation

Severity: High	Category: Business Logic
Target: - contracts/EarnManager.sol	

Description

The `EarnManager.rebalance` function logic relies on the assumption that decreasing `position.liquidity` will fully deplete the liquidity within the associated PancakeSwap V3 Position NFT, allowing it to be burned. However, the `NonfungiblePositionManager` (NPM) allows any user to add liquidity to an existing NFT via its `increaseLiquidity` function.

A malicious actor can "donate" a negligible amount of liquidity (e.g., 1 wei) directly to the `targetNFT` handled by `EarnManager`. This creates a discrepancy where the NFT's actual liquidity exceeds the amount tracked or processed by the `rebalance` function. Since PancakeSwap's `burn` operation requires the position's liquidity to be exactly zero, these "dust" deposits will cause the `burn` call to revert, effectively bricking the rebalancing mechanism for that position.

contracts/EarnManager.sol:L381-L443

```
function rebalanceLiquidity(
    int24 tickLower,
    int24 tickUpper,
    uint24 fee
) external onlyOwner {
    (uint256 amount0, uint256 amount1) = _decreaseLiquidity(position.liquidity);

    if (isEarningCAKE) {
        pcsMasterChef.burn(position.tokenId);
    } else {
        pcsNfpManager.burn(position.tokenId);
    }
    ...
}
```

Recommendation

Fetch and use the actual liquidity in function `rebalanceLiquidity`.

Status

This issue has been resolved by the team with commit 5db3be2.

2. Ineffective Slippage Protection in rebalanceLiquidity

Severity: High	Category: Business Logic
Target: - contracts/EarnManager.sol	

Description

The `EarnManager.rebalanceLiquidity` function implements a slippage protection mechanism that is fundamentally flawed. The function relies on `_getInputAmounts` to calculate expected entry/exit values; however, this internal calculation derives its pricing data directly from the current spot price of the liquidity pool.

Since the "expected" amounts are calculated based on the pool's state at the moment of execution, an attacker can **front-run** the rebalance transaction using a large swap or flash loan to skew the pool's price. Consequently, `_getInputAmounts` will produce "acceptable" slippage limits that align with the manipulated price. This renders the slippage check ineffective, as the "protection" dynamically adjusts to favor the attacker's manipulated state.

contracts/EarnManager.sol:L381-L443

```
function rebalanceLiquidity(
    int24 tickLower,
    int24 tickUpper,
    uint24 fee
) external onlyOwner {
    (uint256 amount0Input, uint256 amount1Input) = _getInputAmounts(position.token0,
position.token1, amount0, amount1);
    uint256 amount0Min = amount0Input * (ONE - slippage) / ONE;
    uint256 amount1Min = amount1Input * (ONE - slippage) / ONE;
    ...
}
```

Recommendation

Add one expected sqrt price parameter. This will make sure that most assets will be re-added into the liquidity pool.

Status

This issue has been resolved by the team with commit 5db3be2.

3. Users may fail to add liquidity

Severity: High	Category: Business Logic

Target:
- contracts/EarnManager.sol

Description

In EarnManager, users can deposit underlying assets to earn profit. The actual input amount will be calculated via function `_getInputAmounts`.

The problem here is that if the current price is larger than the upper price, increased liquidity will always fail because of improper process in function `getAmount1FromAmount0`.

contracts/EarnManager.sol:L284-L330

```
function increaseLiquidity(
    uint256 amount0Desired,
    uint256 amount1Desired
) external whenNotPaused nonReentrant {
    (uint256 amount0Input, uint256 amount1Input)
        = _getInputAmounts(position.token0, position.token1, amount0Desired,
amount1Desired);
    (uint128 newLiquidity,,) = _increaseLiquidity(amount0Input, amount1Input);
    ...
}
```

contracts/EarnManager.sol:L530-L549

```
function _getInputAmounts(
    address _token0, address _token1, uint256 _amount0Desired, uint256 _amount1Desired
) internal view returns (uint256 amount0Input, uint256 amount1Input) {
    amount1Input = earnHelper.getTokenXInputAmount(_token0, _amount0Desired);
    ...
}
```

contracts/EarnHelper.sol:L353-L376

```
function getAmount1FromAmount0(
    address pool, uint256 amount0Desired, int24 tickLower, int24 tickUpper
) public view returns (uint256 amount1Needed) {
   ...
   if (sqrtPriceX96 <= sqrtRatioAX96) {
       return 0;
   } else if (sqrtPriceX96 >= sqrtRatioBX96) {
       revert("Price above range, should use getAmount0FromAmount1");
   }
}
```

Recommendation

Refactor the function `getAmount1FromAmount0`.

Status

This issue has been resolved by the team with commit 5db3be2.

4. Stale `pcsPool` Reference Following Pool Fee Configuration Update

Severity: Medium	Category: Business Logic
Target: - contracts/EarnManager.sol	

Description

The `EarnManager.rebalanceLiquidity` function allows the protocol owner to adjust the pool `fee` during a liquidity migration. While the logic successfully shifts liquidity between different fee tiers, the contract fails to update the internal `pcsPool` state variable to reflect the new pool's address.

In PancakeSwap V3, pools are uniquely identified by the tuple `(token0, token1, fee)`. Changing the `fee` implicitly targets an entirely different smart contract. By maintaining a reference to the "stale" `pcsPool`, all subsequent operations—including price fetching, oracle queries, and liquidity accounting—will be performed against the old pool, leading to critical calculation errors and potential fund mismanagement.

contracts/EarnManager.sol:L381-L443

```
function rebalanceLiquidity(
    int24 tickLower,
    int24 tickUpper,
    uint24 fee
) external onlyOwner {
    // 3. add to pcs with new tickLower, tickUpper and fee
    (uint256 amount0Input, uint256 amount1Input) = _getInputAmounts(position.token0,
position.token1, amount0, amount1);
}
```

Recommendation

Update the `pcsPool` timely in `rebalanceLiquidity`.

Status

This issue has been resolved by the team with commit 5db3be2.

5. Centralization risk

Severity: Medium	Category: Business Logic
Target: - contracts/EarnManager.sol	

Description

In the EarnManager contract, there exists one privileged role, `owner`. The privileged role has the authority to execute some key functions such as `upgradeTo`, `sweepToken`, etc.

If these roles' private keys are compromised, an attacker could trigger these functions to steal the rewards.

contracts/EarnManager.sol:L509-L523

```
function sweepToken(address token, uint256 amount, address receiver) external onlyOwner
{
    require(token != address(0), "Invalid token address");
    require(amount > 0, "Invalid amount");

    if (token == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) {
        require(address(this).balance >= amount, "Insufficient balance");
        (bool success, bytes memory data) = receiver.call{value: amount}("");
        require(success, string(data));
    } else {
        require(IERC20(token).balanceOf(address(this)) >= amount, "Insufficient
balance");
        SafeERC20.safeTransfer(IERC20(token), receiver, amount);
    }

    emit SweepToken(token, amount, receiver);
}
```

Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.\

Status

This issue has been mitigated by the team.

6. Deadline check does not work

Severity: Low	Category: Business Logic

Target:
- contracts/EarnManager.sol

Description

Within the `EarnManager` contract, functions for adjusting liquidity (increase/decrease) utilize a hardcoded deadline calculated as `block.timestamp + 10`. While intended as a safety buffer, this implementation effectively bypasses the deadline protection mechanism. Since `block.timestamp` is only determined at the moment of block execution, the deadline will always be "now + 10 seconds" relative to the miner's clock, making it impossible for a transaction to expire while sitting in the mempool.

contracts/EarnManager.sol:L583-L621

```
function _decreaseLiquidity(uint128 liquidity) internal returns (uint256 amount0,
uint256 amount1) {
    if (isEarningCAKE) {
        // 3. decrease liquidity from MasterChefV3
        (amount0, amount1) =
pcsMasterChef.decreaseLiquidity(INonfungiblePositionManagerStruct.DecreaseLiquidityParam
s({
            tokenId: position.tokenId,
            liquidity: liquidity,
            amount0Min: amount0Min,
            amount1Min: amount1Min,
            deadline: block.timestamp + 10
        }));
    }
}
```

Recommendation

Suggest adding `deadline` parameter.

Status

This issue has been mitigated by the team.

7. Missing input validation in mint

Severity: Low	Category: Business Logic
Target: - contracts/EarnManager.sol	

Description

The `EarnManager` protocol relies on a pre-initialized `pcsPool` address set via `setPancakeSwapContracts`. However, the `mint` function allows the owner to manually input a `fee` parameter when creating the initial liquidity position.

If the provided `fee` does not correspond to the fee tier of the stored `pcsPool`, the `mint` operation will interact with a different PancakeSwap V3 pool than the one tracked by the contract. This creates a "split state" where the contract's internal logic (such as price oracles, fee collection, and rebalancing) queries the `pcsPool` address, while the protocol's actual assets are locked in a different, untracked pool contract.

contracts/EarnManager.sol:L217-L273

```
function mint(
    uint256 amount0Desired,
    uint256 amount1Desired,
    int24 tickLower,
    int24 tickUpper,
    uint24 fee
) external onlyOwner {
    require(amount0Desired > 0 && amount1Desired > 0, "Invalid amount");
    // 0. check if the position is already minted
    require(position.tokenId == 0, "Position already minted");
    // set the tick lower and tick upper.
    position.tickLower = tickLower;
    position.tickUpper = tickUpper;
    position.fee = fee;
}
```

Recommendation

Add one input validation and make sure that the `pcsPool`'s fee matches `fee`.

Status

This issue has been resolved by the team with commit 5db3be2.

8. Incorrect current price calculation

Severity: Low	Category: Business Logic
Target: - contracts/EarnHelper.sol	

Description

In EarnHelper, the function `getCurrentPrice` is used to calculate token0/token1 price.

The assumption is that `_getPriceToken1ToToken0` returns `1e18 * token0 / token1`. However, the return value equals `1e18 * token1 / token0`. This will cause the price calculation to be incorrect.

contracts/EarnHelper.sol:L145-L166

```
function getCurrentPrice(address fromToken, address token0, address token1) public view
returns (uint256 price) {
    uint8 token0Decimals = IERC20Metadata(token0).decimals();
    uint8 token1Decimals = IERC20Metadata(token1).decimals();

    if (fromToken == token0) {
        price = _getPriceToken0ToToken1(earnManager.pcsPool());
        price = FullMath.mulDiv(price, 10 ** token0Decimals, 10 ** token1Decimals);
    } else if (fromToken == token1) {
        // Get price: 1 token1 = ? token0 (scaled by 1e18)
        // _getPriceToken1ToToken0 returns: 1e18 * token0/token1
        price = _getPriceToken1ToToken0(earnManager.pcsPool());
        price = FullMath.mulDiv(price, 10 ** token1Decimals, 10 ** token0Decimals);
    } else {
        revert("Invalid token");
    }
}
```

Recommendation

Calculate the token0/token1 price correctly.

Status

This issue has been resolved by the team with commit 5db3be2.

2.3 Informational Findings

9. Redundant code

Severity: Informational	Category: Redundancy
Target: - contracts/EarnManager.sol	

Description

The `EarnManager.setSlippage` function implements a security check `require(_slippageBps >= 0)`. However, since the variable `_slippageBps` is defined as an unsigned integer `(uint256)`, it is mathematically impossible for its value to be less than zero.

By the definition of unsigned types in Solidity, this condition will always evaluate to `true`. Consequently, the intended security layer—presumably meant to prevent invalid or negative input—is functionally non-existent, providing a false sense of security while consuming unnecessary gas.

contracts/EarnManager.sol:L162-L167

```
function setSlippage(uint256 _slippageBps) external onlyOwner {
    require(_slippageBps >= 0 && _slippageBps <= MAX_SLIPPAGE, "Invalid slippage BPS");
    slippage = _slippageBps;

    emit SetSlippage(_slippageBps);
}
```

Recommendation

Remove unnecessary code.

Status

This issue has been resolved by the team with commit 5db3be2.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 0591e96:

File	SHA-1 hash
EarnManager.sol	1d0d467e19a5a99391f97a1072ce9f499ae8f8b0
EarnHelper.sol	db882d958c155cb8c0fc305326213dc98f79bea1