SALUS SECURITY

JAN 2026

CODE SECURITY ASSESSMENT

NIZA GLOBAL

Overview

Project Summary

- Name: Nizaglobal - Perma dex
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
  - https://github.com/nizaglobal/perma_dex_swap
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Nizaglobal - Perma dex
Version	v3
Type	Solidity
Dates	Jan 20 2026
Logs	Jan 16 2026; Jan 20 2026; Jan 20 2026

Vulnerability Summary

Total High-Severity issues	4
Total Medium-Severity issues	0
Total Low-Severity issues	1
Total informational issues	2
Total	7

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 issue

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Anyone can withdraw liquidity from FeeManager-Owned Uniswap v4 positions	High	Business Logic	Resolved
2	Dos due to missing sync() before settle()	High	Denial of Service	Resolved
3	Delta handle error	High	Denial of Service	Resolved
4	The same position‘s salt will be overwritten	High	Business Logic	Resolved
5	Missing Test Suite for UniswapV4 function	Low	Testing	Acknowledged
6	Missing events for swapV4ExactInputSingle function	Informational	Logging	Resolved
7	Use of floating pragma	Informational	Configuration	Resolved

2.2 Notable Findings

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

1. Anyone can withdraw liquidity from FeeManager-Owned Uniswap v4 positions	
Severity: High	Category: Business Logic
Target: - src/FeeManager.sol	

Description

`FeeManager` adds/removes Uniswap V4 liquidity via `PoolManager.unlock()` and `unlockCallback()`. In Uniswap V4, positions are keyed by `(poolId, owner, tickLower, tickUpper, salt)` and owner is `msg.sender` of `PoolManager.modifyLiquidity`.

Because `FeeManager.unlockCallback()` is the component that calls `poolManagerV4.modifyLiquidity()`, the owner recorded by V4 for all liquidity operations is the `FeeManager` contract, not the user. Additionally, `salt` is hardcoded to `bytes32(0)`, causing all liquidity for the same `(PoolKey, tickLower, tickUpper)` to be aggregated into a single contract-owned position.

src/FeeManager.sol:L426-L448

```
function _executeLiquidityModification(...) internal returns (BalanceDelta, BalanceDelta)
{
    Return abi.decode(poolManagerV4.unlock(abi.encode(isAdd? CallbackType.AddLiquidity :
CallbackType.RemoveLiquidity, key, tickLower, tickUpper, liquidityDelta)),
(BalanceDelta, BalanceDelta));
}
```

@uniswap/v4-core/src/PoolManager.sol:L143-L170

```
/// @inheritdoc IPoolManager
   function modifyLiquidity(
       PoolKey memory key,
       IPoolManager.ModifyLiquidityParams memory params,
       bytes calldata hookData
   ) external onlyWhenUnlocked noDelegateCall returns (BalanceDelta callerDelta,
BalanceDelta feesAccrued) {
       ...
           (principalDelta, feesAccrued) = pool.modifyLiquidity(
               Pool.ModifyLiquidityParams({
                   owner: msg.sender,
                   tickLower: params.tickLower,
                   tickUpper: params.tickUpper,
                   liquidityDelta: params.liquidityDelta.toInt128(),
                   tickSpacing: key.tickSpacing,
                   salt: params.salt
            })
           );
           // fee delta and principal delta are both accrued to the caller
```

```
            callerDelta = principalDelta + feesAccrued;
}
```

`removeLiquidityV4()` is permissionless and does not track/verify user ownership or shares. Any caller can invoke `removeLiquidityV4()` with a negative `liquidityDelta`, which reduces the contract-owned position during the unlock callback. The withdrawn tokens are then transferred unconditionally to `msg.sender`.

src/FeeManager.sol:L457-L496

```
function removeLiquidityV4(
   PoolKey calldata key,
   int24 tickLower,
   int24 tickUpper,
   int256 liquidityDelta
) external nonReentrant returns (uint256 amount0, uint256 amount1) {
   …
   IERC20(Currency.unwrap(key.currency0)).safeTransfer(
       msg.sender,
       amount0
   );
   IERC20(Currency.unwrap(key.currency1)).safeTransfer(
       msg.sender,
       amount1
   );
   …
}
```

Any user's liquidity deposited through `addLiquidityV4()` can be removed by an arbitrary third party, who will receive the withdrawn `token0/token1`. This enables complete theft of funds from all liquidity aggregated under the contract-owned position.

Recommendation

Implement explicit `ownership/shares` accounting for V4 liquidity positions, and enforce it on withdrawal.

Status

The team has resolved this issue in commit 35de79e.

2. Dos due to missing sync() before settle()

Severity: High	Category: Denial of Service
Target: - src/FeeManager.sol	

Description

Uniswap V4's `PoolManager` uses a "flash accounting" settlement model during `unlock()`. After `swap()/modifyLiquidity()`, the caller accrues per-currency deltas, and must fully settle them before `unlock()` returns, otherwise `PoolManager.unlock()` reverts.

In V4, paying an owed ERC20 delta requires the sequence:

1. Sync the currency balance with `sync()`
2. Transfer the tokens to the `PoolManager`
3. Complete the settlement with `settle()`

However, `FeeManager` settles positive deltas (`delta.amountX() > 0`) by doing `transfer + settle` without calling `sync()`.

src/FeeManager.sol:L551-L623

```
function _settleSwapDeltas(
    PoolKey memory key,
    BalanceDelta delta
) internal {
    if (delta.amount0() > 0) {
        IERC20(Currency.unwrap(key.currency0)).safeTransfer(...
        poolManagerV4.settle();
    }
    ...
}

function _settleLiquidityDeltas(
    PoolKey memory key,
    BalanceDelta delta
) internal {
    if (delta.amount0() > 0) {
        IERC20(Currency.unwrap(key.currency0)).safeTransfer(...
        );
        poolManagerV4.settle();
    }
    ...
}
```

Recommendation

Fix the implementation to align with the documentation.

Status

The team has resolved this issue in commit 35de79e.

3. Delta handle error

Severity: High	Category: Denial of Service
Target: - src/FeeManager.sol	

Description

The Uniswap V4 guides indicates that:

- settle - used following token transfers to the manager or burning of ERC6909 claims to resolve negative deltas
- take - transfer tokens from the manager, used to resolve positive deltas but also provide token loans, producing negative deltas

Analysis of the code reveals that V4 BalanceDelta semantics are inversely applied: positive delta signifies the contract should `take()` from the PoolManager, while negative delta signifies the contract should `settle()`. The current implementation erroneously uses positive delta to `settle()` and negative delta to `take`, resulting in completely inverted behavior.

src/FeeManager.sol:L282-L311

```
function _settleSwapDeltas(
    PoolKey memory key,
    BalanceDelta delta
) internal {
    if (delta.amount0() > 0) {
        IERC20(Currency.unwrap(key.currency0)).safeTransfer(
            address(poolManagerV4),
            uint256(int256(delta.amount0()))
        );
        poolManagerV4.settle();
    }

    if (delta.amount1() > 0) {
        IERC20(Currency.unwrap(key.currency1)).safeTransfer(
            address(poolManagerV4),
            uint256(int256(delta.amount1()))
        );
        poolManagerV4.settle();
    }

    if (delta.amount0() < 0) {
        poolManagerV4.take(
            key.currency0,
            address(this),
            uint256(int256(-delta.amount0()))
        );
    }

    if (delta.amount1() < 0) {
        poolManagerV4.take(
            key.currency1,
            address(this),
            uint256(int256(-delta.amount1()))
        );
    }
}

function _settleLiquidityDeltas(
    PoolKey memory key,
    BalanceDelta delta
) internal {
    if (delta.amount0() > 0) {
        IERC20(Currency.unwrap(key.currency0)).safeTransfer(
            address(poolManagerV4),
            uint256(int256(delta.amount0()))
        );
        poolManagerV4.settle();
    }

    if (delta.amount1() > 0) {
        IERC20(Currency.unwrap(key.currency1)).safeTransfer(
            address(poolManagerV4),
            uint256(int256(delta.amount1()))
        );
        poolManagerV4.settle();
    }

    if (delta.amount0() < 0) {
        poolManagerV4.take(
            key.currency0,
            address(this),
            uint256(int256(-delta.amount0()))
        );
    }

    if (delta.amount1() < 0) {
        poolManagerV4.take(
            key.currency1,
            address(this),
            uint256(int256(-delta.amount1()))
        );
    }
}
```

The V4 swap and liquidity operations can be subjected to DoS attacks, with settlement logic potentially resulting in asset loss or fund lockup.

Recommendation

Fix the implementation to align with the documentation.

Status

The team has resolved this issue in commit 35de79e.

4. The same position‘s salt will be overwritten	
Severity: High	Category: Business Logic
Target: - src/FeeManager.sol	

Description

The `addLiquidityV4` function is responsible for adding user liquidity to Uniswap V4 pools while generating and recording a unique salt for each user's position defined by `(key, tickLower, tickUpper)`. This salt enables authorization verification when `removeLiquidityV4` is later called. The `removeLiquidityV4` function validates the position holder using the stored salt before executing the liquidity removal.

src/FeeManager.sol:L376-L456

```
function addLiquidityV4(...)
    external
    nonReentrant
    returns (uint256 amount0, uint256 amount1, bytes32 salt)
{
    ...
    bytes32 positionKey = _getPositionKey(key, tickLower, tickUpper);
    salt = keccak256(
        abi.encodePacked(msg.sender, userNonces[msg.sender]++)
    );
    userPositionSalts[msg.sender][positionKey] = salt;

    ...
}
function _getPositionKey(
    PoolKey calldata key,
    int24 tickLower,
    int24 tickUpper
) internal pure returns (bytes32) {
    return keccak256(abi.encode(key, tickLower, tickUpper));
}
function removeLiquidityV4(
    PoolKey calldata key,
    int24 tickLower,
    int24 tickUpper,
    int256 liquidityDelta,
    bytes32 salt
) external nonReentrant returns (uint256 amount0, uint256 amount1) {
    …
    bytes32 positionKey = _getPositionKey(key, tickLower, tickUpper);
    require(
        userPositionSalts[msg.sender][positionKey] == salt,
        "Unauthorized: not your position"
    );
…
```

The `positionKey` consists only of `(key, tickLower, tickUpper)`. Each `addLiquidityV4` invocation overwrites the existing `userPositionSalts[msg.sender][positionKey]` value, resulting in permanent loss of the previous salt.

Recommendation

Change to a `salt -> owner` mapping and allow removal with any historical salt.

Status

The team has resolved this issue in commit 8adc8b2.

5. Missing Test Suite for UniswapV4 function	
Severity: Low	Category: Testing
Target: - src/FeeManager.sol	

Description

A test suite to confirm the proper operation of the deployed contracts and crucial features like registering and admin role assignment is absent from the repository. Contract security and dependability are at risk because defects or regressions might be undiscovered. Current test coverage does not include UniswapV4-related features.

Recommendation

Add integration tests based on V4 official test components.

Status

This issue has been acknowledged by the team.

2.3 Informational Findings

6. Missing events for swapV4ExactInputSingle function	
Severity: Informational	Category: Logging
Target: - src/FeeManager.sol	

Description

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes that allow users to evaluate them. Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in protocol users.

In the `FeeManager`contract, events are lacking in the privileged functions (e.g. `swapV4ExactInputSingle()`).

Recommendation

It is recommended to emit events for critical state changes.

Status

The team has resolved this issue in commit 35de79e.

7. Use of floating pragma	
Severity: Informational	Category: Configuration
Target: - All	

Description

```
pragma solidity ^0.8.24;
```

For example, the `FeeManager` uses a floating compiler version ^0.8.24.

Using a floating pragma ^0.8.24 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

The team has resolved this issue in commit 35de79e.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 1758396:

File	SHA-1 hash
FeeManager.sol	b1da13d3f92b7f4b8922d08cc910bc514fe3b353