SALUS SECURITY

JUL 2025

CODE SECURITY ASSESSMENT

VANILLA FINANCE

Overview

Project Summary

- Name: Vanilla Finance - MemePerps
- Platform: The BSC Blockchain
- Language: Solidity
- Repository:
    - https://github.com/VanillaDevTeam/MemePerps
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Vanilla Finance - MemePerps
Version	v4
Type	Solidity
Dates	Aug 02 2025
Logs	Jul 18 2025; Jul 22 2025; Jul 23 2025; Aug 02 2025

Vulnerability Summary

Total High-Severity issues	4
Total Medium-Severity issues	4
Total Low-Severity issues	4
Total informational issues	3
Total	15

Contact

E-mail: support@salusec.io

SALUS

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.

SALUS

Content

SALUS

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.

SALUS

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Signature not bound to parameters enables forged and replay Attacks	High	Business Logic	Resolved
2	Unvalidated params enables bad debt and draining of protocol-held balance	High	Data Validation	Mitigated
3	The whitelist pool will be drained	High	Business logic	Resolved
4	Liquidity manipulation attack	High	Business logic	Mitigated
5	Inflation attack	Medium	Business Logic	Resolved
6	Pools remain active after token de-whitelisting, with no pause/close mechanism	Medium	Configuration	Resolved
7	Lack of slippage check in closePositionWithData	Medium	Data Validation	Resolved
8	Centralization risk	Medium	Centralization	Resolved
9	Create pool can be front-runing	Low	Front-running	Resolved
10	Unit mismatch in getQuoterAmountIn	Low	Numerics	Resolved
11	Inconsistent decimal handling	Low	Inconsistency	Resolved
12	Use a strict less than sign in getAllPositions	Low	Business Logic	Resolved
13	Gas optimization suggestions	Informational	Gas optimization	Resolved
14	Use of floating pragma	Informational	Configuration	Resolved
15	Callback adaptation error	Informational	Business Logic	Resolved

SALUS

2.2 Notable Findings

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

1. Signature not bound to parameters enables forged and replay Attacks

Severity: High	Category: Business logic
Target: - contracts/leverage/facets/LeverageTradingOperationsFacet.sol	

Description

The functions `_decodeOpenPositionParams()`, `_decodeClosePositionParams()`, `_decodeLiquidatePositionParams()`, each split the caller-supplied signedData into `(bytes32 message, uint8 v, bytes32 r, bytes32 s, <Params> params)`, verifying only that `(v,r,s)` form a valid signature over the 32-byte message

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L1015-L1088

```
function _decodeOpenPositionParams(
   bytes calldata signedData
) internal view returns (OpenPositionParams memory) {
   (
       bytes32 message,
       uint8 v,
       bytes32 r,
       bytes32 s,
       OpenPositionParams memory params
   ) = abi.decode(
           signedData,
           (bytes32, uint8, bytes32, bytes32, OpenPositionParams)
       );

   // Create Ethereum signed message hash
   bytes32 ethSignedMessageHash = keccak256(
       abi.encodePacked("\x19Ethereum Signed Message:\n32", message)
   );
   address signer = ecrecover(ethSignedMessageHash, v, r, s);
   if (!hasRole(DATA_PROVIDER_ROLE, signer))
       revert Error.Unauthorized();

   return params;
}

function _decodeClosePositionParams(bytes calldata signedData) internal view
returns (ClosePositionParams memory) {...}

function _decodeLiquidatePositionParams(bytes calldata signedData) internal view
returns (LiquidatePositionParams memory) {...}
```

The contract never verifies that the message is derived from, or otherwise commits to, the decoded parameters.

SALUS

As a result, an attacker with access to any previously published signature from a whitelisted `DATA_PROVIDER_ROLE` address can craft malicious signedData packets by:

1. Re-use a valid `(message, v, r, s)` tuple.
2. Append arbitrary `OpenPositionParams`, `ClosePositionParams`, or `LiquidatePositionParams`.
3. Replaying the same signature indefinitely, since there is no nonce or deadline mechanism in place to prevent reuse..

Recommendation

Bind signature to `params`, add replay protection(`nonce`, `deadline`), adopt EIP-712 typed-data signatures so the whole struct is signed in an unforgeable way.

Status

The team has resolved this issue in commit 373f490.

SALUS

2. Unvalidated params enables bad debt and draining of protocol-held balance

Severity: High	Category: Data Validation
Target: - contracts/leverage/facets/LeverageTradingOperationsFacet.sol	

Description

In the `openPositionWithData()`, `closePositionWithData()`, and `liquidatePositionWithData()` functions, the protocol accepts off-chain signed structs (`OpenPositionParams`, `ClosePositionParams`, `LiquidatePositionParams`). However, many of the economically critical fields in these structs are only subjected to basic sanity checks on-chain.

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L148-L179

```
struct OpenPositionParams {
    address marginToken;
    address targetToken;
    uint256 marginAmount;
    uint256 maintenanceMarginRate;
    uint256 borrowAmount;
    uint256 fee;
    uint256 swapAmount;
    uint256 leverageMultiplier;
    uint256 minTargetTokenAmount;
    uint256 slippage;
    uint256 positionId;
    bool longOrShort;
    uint256 deadline;
}

struct ClosePositionParams {
    uint256 positionId;
    uint256 slippage;
    uint256 interest;
    bytes interestHistory;
    uint256 deadline;
}

struct LiquidatePositionParams {
    uint256 positionId;
    uint256 liquidationPenalty;
    uint256 liquidatorRewardRate;
    uint256 totalInterest;
    bytes interestHistory;
    uint256 deadline;
}
```

Coupled with the previously reported issue **“Signature Not Bound to Parameters enables forged and replay Attacks”** (any valid `(v,r,s)` can be reused against arbitrary structs), an attacker can Inject unlimited fees, exaggerate leverageMultiplier, and perform other unauthorized manipulations, for example:

When opening a position, the untrusted `swapAmount` is used at

SALUS

When opening a position, the untrusted `swapAmount` is used in the following critical functions: `Utils.forceApprove(tokenIn, dexRouter, swapAmount)`, `dexRouter.exactInputSingle(... amountIn: swapAmount ...)`,. However, there is no invariant that ties `swapAmount` to the sum of (margin + borrow) for long positions or to the borrowed target amount for short positions. Additionally, there is no balance delta check or cap, and signatures are not bound to parameters (see separate High finding). This allows attackers to supply arbitrary values, exploiting the system.

Attack A – Under-Swap to Manufacture Bad Debt (Insolvency)

An attacker can borrow a large amount but set a minuscule `swapAmount`. Most of the borrowed tokens remain idle in the contract, never being swapped into the position. As a result, the accounting records a large `borrowedAmount` but only a tiny `actualTokenAmount`. Upon position closure, repayment is calculated based on the small traded exposure, leaving the position in NegativeEquity. The lending pool absorbs the shortfall while the borrowed tokens remain stranded in the contract. Repeating this attack can drain the pool's solvency.

Attack B – Over-Swap to Drain Protocol-Held Balances

An attacker can open a small position while supplying a disproportionately large `swapAmount` , potentially up to the contract's full balance. The `_openPosition` function will approve and execute the entire trade, resulting in a large `actualTargetAmount` being attributed to the attacker's position. Upon closing, only the small recorded borrow is repaid, and the excess proceeds are returned to the attacker as `marginReturned`, allowing them to effectively loot protocol-owned or other users' residual funds.

Recommendation

Add on chain validation for these params.

Status

The team has mitigated this issue in commit 2a91761.

3. The whitelist pool will be drained

Severity: High	Category: Business logic
Target: - contracts/leverage/facets/LeverageTradingOperationsFacet.sol	

Description

In the `LeverageTradingOperationsFacet` contract, the `createToken()` function will select the correct fund pool for subsequent borrowing and trading operations according to the direction selected by the user (long/short). But this only checks one token.

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L259-L300

```
function openPositionWithData(...){
    ...
    uint256 poolId;
    if (params.longOrShort) {
        poolId = lts.stakingContract.findPoolByToken(actualMarginToken);
    } else {
        poolId = lts.stakingContract.findPoolByToken(params.targetToken);
    }

    if (poolId == type(uint256).max) revert("Pool not supported");
```

Attach Scenario

1. The protocol adds `USDT` to white list.
2. Attacker builds a `FAKE-USDT` pool with a small supply ratio of 1 `FAKE` : 100 `USDT`.
3. Attacker opens a 10U ten-times leveraged `FAKE` long position. The protocol will swap 100 `USDT` for `FAKE` and the `USDT` supply of the pool will increase to 200 `USDT`.
4. The attacker uses `FAKE` tokens to drain the pool. The profit is 100U, the cost is 10U, resulting in a net gain of 90U

Through the above method, the attacker can drain all whitelisted tokens.

Recommendation

Add a check for ensuring both `marginToken` and `targetToken` are on the whitelist.

Status

The team has resolved this issue in commit 373f490.

SALUS

4. Liquidity manipulation attack

Severity: High	Category: Business logic
Target: - contracts/leverage/PriceFeed.sol	

Description

The contract's `findPool()` function finds the pool with the highest liquidity between tokenA and tokenB across different fee tiers in the PancakeSwap V3 factory and returns its address. This introduces a potential issue where the pool used to open a position differs from the one used to close it, potentially enabling price manipulation.

contracts/leverage/PriceFeed.sol:L82-L106

```
function findPool(...)
{
    address[] memory pools = new address[](4);
    pools[0] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 100);
    pools[1] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 500);
    pools[2] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 2500);
    pools[3] = IPancakeV3Factory(factory).getPool(tokenA, tokenB, 10000);
    //get best liquidity pool with best liquidity
    uint256 bestLiquidity = 0;
    address bestPool = address(0);
    for (uint256 i = 0; i < pools.length; i++) {
        if (pools[i] != address(0)) {
            uint256 liquidity = IPancakeV3Pool(pools[i]).liquidity();
            if (liquidity > bestLiquidity) {
                bestLiquidity = liquidity;
                bestPool = pools[i];
            }
        }
    }

    return bestPool;
}
```

Attach Scenario

1. The `Meme-USDT` has one uninitialized pool and three pools with existing liquidity.
2. Attacker initialize the pool at a high price and open long positions at the same time.
3. By leveraging the flash loan feature of the pools, the attacker combines the liquidity of three pools to create one pool with high liquidity.
4. The attacker closes the position using the manipulated pool for settlement, realizes the profit, and repays the flash loans from the three pools

Recommendation

Add a mechanism to ensure price consistency by using the same pool for both opening and closing positions.

Status

The team has mitigated this issue in commit 373f490.

SALUS

5. Inflation attack

Severity: Medium	Category: Business Logic
Target: - contracts/staking/facets/StackOperationFacet.sol	

Description

The protocol rounds down for `unstake()` and `_stake()` functions. For an empty pool, the malicious user can manipulate it to cause inflation attacks. Specifically, the normal ratio of share to amount will not exceed 1:2.

The attacker also can transfer to the contract directly, creating inconsistency between the contract's internal and recorded balances.

The two ways can lead to Inflation attacks which makes this ratio very large.

contracts/staking/facets/StackOperationFacet.sol:L233-323

```
function unstake(uint256 _pid, uint256 _amount) external returns (uint256) {
    ...
    uint256 balance = _balanceAndBorrowedOfPool(_pid);
    uint256 userShares = _amount.mulDiv(user.shares, user.amount);
    uint256 normalizedRewards = calculateAmount(
        userShares,
        pool.totalShares,
        balance
    );
    ...
}
function _stake(uint256 _pid, uint256 _amount, address _staker) internal {
    ...
    if (pool.totalStaked == 0) {
        shares = normalizedAmount;
    } else {
        shares = calculateShares(
            normalizedAmount,
            totalStaked,
            pool.totalShares
        );
    }
    ...
```

Due to the pool lack of lending attack vector, the severity is medium.

Recommendation

It is recommended to ensure that markets are never empty by minting small share (or equivalent) balances at the time of pool creation, preventing the rounding error being used maliciously. Or use one-to-one minting when `totalShares` and `totalStaked` are at smaller values.

Status

The team has resolved this issue in commit 373f490.

SALUS

6. Pools remain active after token de-whitelisting, with no pause/close mechanism

Severity: Medium	Category: Configuration
Target: - contracts/staking/facets/StackOperationFacet.sol	

Description

`StackOperationFacet.batchRemoveFromWhitelist()` removes a token from the `whitelistTokensArray`, but does not touch the associated pool state:

1. `s.pools[pid].isActive` is left true.
2. `s.tokenToPoolId[_token]` and `s.poolExists[_token]` remain set.
3. `s.creatorPools[...]` still references the pool.
4. No call path marks the pool disabled or prevents subsequent use.

As a result, after governance removes a token, users can continue to route leverage trades through the stale pool.

Recommendation

Update pool state variables when de-whitelisting a token and add checks in stake, unstake, borrow, and repay, e.g. `require(pool.isActive && isTokenWhitelisted(pool.token))`.

Status

The team has resolved this issue in commit 373f490.

SALUS

7. Lack of slippage check in closePositionWithData

Severity: Medium	Category: Data Validation

Target:
- contracts/staking/facets/StackOperationFacet.sol

Description

The `ClosePositionParams` struct includes a `slippage` field, but `closePositionWithData()` never uses it to constrain the on-chain swap:

contracts/staking/facets/StackOperationFacet.sol:L164-L170

```
    struct ClosePositionParams {
        uint256 positionId;
        uint256 slippage;
        uint256 interest;
        bytes interestHistory;
        uint256 deadline;
    }
```

contracts/staking/facets/StackOperationFacet.sol:L480-L642

```
function closePositionWithData(bytes calldata data){
            if (position.openInfo.longOrShort) {
                allMarginAmount = lts.dexRouter.exactInputSingle(
                    ISwapRouter.ExactInputSingleParams({
                        tokenIn: tokenIn,
                        tokenOut: tokenOut,
                        fee: fee,
                        recipient: address(this),
                        amountIn: swapAmount,
                        amountOutMinimum: 0,
                        sqrtPriceLimitX96: 0,
                        deadline: params.deadline
                    })
                );
            } else {
                allMarginAmount = lts.dexRouter.exactOutputSingle(
                    ISwapRouter.ExactOutputSingleParams({
                        tokenIn: tokenIn,
                        tokenOut: tokenOut,
                        fee: fee,
                        recipient: address(this),
                        amountOut: swapAmount,
                        amountInMaximum: type(uint256).max,
                        sqrtPriceLimitX96: 0,
                        deadline: params.deadline
```

Recommendation

Add slippage checks in the `closePositionWithData()` function.

Status

The team has resolved this issue in commit 373f490.

SALUS

8. Centralization risk

Severity: Medium	Category: Centralization
Target: - contracts/staking/facets/StackOperationFacet.sol	

Description

The `StackOperationFacet` contract has privileged accounts. These privileged accounts can borrow all tokens from the contracts without any collateral by using the `borrow()` function, and change any user's share by using the `updateUserShares()`` functions.

If privileged accounts' private key or admin's is compromised, an attacker can steal all the tokens in the contract.

If the privileged accounts are plain EOA accounts, this can be worrisome and pose a risk to the other users.

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

The team has resolved this issue in commit 373f490 and state that a multi-signature wallet will be used on the official network.

SALUS

9. Create pool can be front-runing

Severity: Low	Category: Front-running

Target:
- contracts/staking/facets/StackOperationFacet.sol

Description

In the StackOperationFacet contract, once the contract owner calls `addTokenToWhitelist()` function:

contracts/staking/facets/StackOperationFacet.sol:L102-L104

```
function addTokenToWhitelist(address _token) external onlyOwner {
    _addTokenToWhitelistArray(_token);
}
```

An attacker can monitor the mempool for the transaction and then call the `stakeByToken()` function, front-run the subsequently expected `createPool()` call by the contract owner.

contracts/staking/facets/StackOperationFacet.sol:L201-L225

```
function stakeByToken(
    address _token,
    uint256 _amount
) external returns (uint256) {
    require(_amount > 0, "Amount must be greater than 0");
    if (_token == address(0)) {
        _token = Constants.WETH_ADDRESS;
    }
    require(
        IStackQueryFacet(address(this)).isTokenWhitelisted(_token),
        "Token not whitelisted"
    );
    uint256 pid = IStackQueryFacet(address(this)).findPoolByToken(_token);

    if (pid == type(uint256).max) {
        pid = _createPool(_token, msg.sender);
    }
    _stake(pid, _amount, msg.sender);
    return pid;
}
```

Therefore, the attacker will become the pool creator, although there are no privileges for creator.

Recommendation

Add an `onlyOwner` modifier on the internal `_createPool()`, or restrict the create pool path in `stakeByToken()`.

Status

The team has resolved this issue in commit 373f490.

SALUS

10. Unit mismatch in getQuoterAmountIn

Severity: Low	Category: Numerics
Target: - contracts/leverage/facets/LeverageTradingQueryFacet.sol	

Description

`getQuoterAmountIn()` queries `IQuoter.quoteExactOutputSingle` with `tokenIn = baseToken` and `tokenOut = quoteToken`. The quoter returns the raw `amountIn` denominated in `baseToken` decimals. However, the code normalizes this value using `quoteToken` decimals:

contracts/leverage/facets/LeverageTradingQueryFacet.sol:L163-L202

```
function getQuoterAmountIn(
    address baseToken,
    address quoteToken,
    uint256 amountOut
) external returns (uint256 amountIn) {
    ...
    (amountIn, , , ) = IQuoter(lts.quoter).quoteExactOutputSingle(
        IQuoter.QuoteExactOutputSingleParams({
            tokenIn: baseToken,
            tokenOut: quoteToken,
            fee: fee,
            amount: Utils.denormalizeTokenAmount(quoteToken, amountOut),
            sqrtPriceLimitX96: 0
        })
    );

    amountIn = Utils.normalizeTokenAmount(quoteToken, amountIn);

    return amountIn;
}
```

If the two tokens have different decimal counts (e.g., 18 vs 6), the normalized result is scaled incorrectly.

Recommendation

Normalize the returned amountIn using `baseToken` decimals.

Status

The team has resolved this issue in commit 373f490 and state that the function is called externally, so the original precision of the currency is retained.

SALUS

11. Inconsistent decimal handling

Severity: Low	Category: Inconsistency
Target: - contracts/leverage/facets/LeverageTradingQueryFacet.sol	

Description

`getQuoterAmountOut()` is intended to return how much `baseToken` you would receive for supplying `amountIn` units of `quoteToken`. However, there is no decimal normalization or denormalization.

contracts/leverage/facets/LeverageTradingQueryFacet.sol:L113-L161

```
function getQuoterAmountOut(
    address baseToken,
    address quoteToken,
    uint256 amountIn
) external returns (uint256) {
    ...
    (uint256 amountOut, , , ) = IQuoter(lts.quoter).quoteExactInputSingle(
        IQuoter.QuoteExactInputSingleParams({
            tokenIn: quoteToken,
            tokenOut: baseToken,
            fee: fee,
            amountIn: amountIn,
            sqrtPriceLimitX96: 0
        })
    );
    Utils.clearAllowance(
        IERC20(quoteToken),
        address(lts.quoter)
    );
    return amountOut;
}
```

Unlike `getQuoterAmountIn()`, which at least attempts to denormalize/normalize units, this function passes amountIn straight through to the quoter.

Additionally, these functions should be read-only; approving tokens for a third-party contract just to compute a quote is unnecessary and dangerous.

Recommendation

Remove all token approvals from quote functions, and add the normalization and denormalization.

Status

The team has resolved this issue in commit 373f490 and state that the function is called externally, so the original precision of the currency is retained.

SALUS

12. Use a strict less than sign in getAllPositions

Severity: Low	Category: Business Logic

Target:
- contracts/leverage/facets/LeverageTradingQueryFacet.sol

Description

Loop excludes `endIndex` because of `<` instead of `<=`:

contracts/leverage/facets/LeverageTradingQueryFacet.sol:L260-L291

```
function getAllPositions(uint256 startIndex, uint256 endIndex) external view
    returns (LibLeverageTradingStorage.Position[] memory positions)
{
    ...
    if (endIndex >= lts.positionIds.length()) {
        endIndex = lts.positionIds.length() - 1;
    }

    uint256 positionsCount = endIndex - startIndex + 1;
    positions = new LibLeverageTradingStorage.Position[](positionsCount);

    for (uint256 i = startIndex; i < endIndex; i++) {
        positions[i] = lts.positions[lts.positionIds.at(i)];
    }
    return positions;
}
```

Recommendation

Use `<=` instead of `<`.

Status

The team has resolved this issue in commit 373f490.

SALUS

2.3 Informational Findings

13. Gas optimization suggestions

Severity: Informational	Category: Gas Optimization

Target:
- contracts/staking/facets/StackOperationFacet.sol
- contracts/leverage/PriceFeed.sol

Description

Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective. Cache array length outside of the loop can save gas.

contracts/staking/facets/StackOperationFacet.sol:L115

```
for (uint256 i = 0; i < _tokens.length; i++) {
```

contracts/staking/facets/StackOperationFacet.sol:L129

```
for (uint256 i = 0; i < _tokens.length; i++) {
```

contracts/leverage/PriceFeed.sol:L96

```
for (uint256 i = 0; i < pools.length; i++) {
```

In the `closePositionWithData()`, the code `position.closeInfo.isSet = true` is executed twice, and should be removed.

contracts/staking/facets/StackOperationFacet.sol:L543-L749

```
function closePositionWithData(bytes calldata data) external nonReentrant
   returns (uint256 marginReturned, int256 realizedProfitLoss){
   ...
   position.closeInfo.isSet = true;
   ...
   position.closeInfo.isSet = true;
   ...
}
```

Recommendation

Consider using the above suggestions to save gas.

Status

The team has resolved this issue in commit 373f490.

SALUS

14. Use of floating pragma

Severity: Informational	Category: Configuration
Target: - All	

Description

```
pragma solidity ^0.8.28;
```

The QuillToken uses a floating compiler version ^0.8.28.

Using a floating pragma ^0.8.28 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 373f490.

SALUS

15. Callback adaptation error

Severity: Informational	Category: Business Logic
Target: - contracts/leverage/facets/LeverageTradingOperationsFacet.sol	

Description

During position opening, closing, and liquidation operations, the protocol performs token swaps with a call flow of `V3SwapRouter.exactOutputSingle -> pool.swap -> V3SwapRouter.pancakeV3SwapCallback`, without triggering any `LeverageTradingOperationsFacet` contract's callback functions. This same situation also applies to Uniswap V3.

contracts/leverage/facets/LeverageTradingOperationsFacet.sol:L998-L1013

```
function pancakeV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata data
) external {
    PancakeHelper.pancakeV3SwapCallback(amount0Delta, amount1Delta, data);
}

// ISwapCallback implementation
function uniswapV3SwapCallback(
    int256 amount0Delta,
    int256 amount1Delta,
    bytes calldata data
) external {
    PancakeHelper.pancakeV3SwapCallback(amount0Delta, amount1Delta, data);
}
```

Recommendation

Remove the Callback function.

Status

The team has resolved this issue in commit 373f490.

SALUS

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit 60818ee:

File	SHA-1 hash
contracts/Constants.sol	5a269719002f73d14291ef83ab700dd2c27a5ea4
contracts/Utils.sol	84e7d7d7d4eff1e729f7346d34c4d4615928d4d6
contracts/leverage/LeverageTradingDiamond.sol	699822fc1feb5fd0da6fc59f5c9e06cec1559efd
contracts/leverage/LeverageTradingInit.sol	3b9899b0374a463e3fb2a8e88b6450403e1b97c1
contracts/leverage/LibLeverageTradingStorage.sol	0ef11b42b8a2d06c6644c2890bf997e207906e2f
contracts/leverage/PancakeHelper.sol	8119ea74608e59b4edb62027e4a4e6f6f4fe1d7c
contracts/leverage/PriceFeed.sol	0a28febf9a360dc3bed054903847881199df9769
contracts/leverage/facets/LeverageTradingQueryFacet.sol	6bb8940cbb123bc3a08b920e25192ce9f14f8e5e
contracts/leverage/facets/LeverageTradingOperationsFacet.sol	ff68aeca4125c3aaf9771af83033c0dd10744851
contracts/staking/MultiTokenStakingDiamond.sol	3bc0bfe2ac8e18210c271b625685ccd0b4217916
contracts/staking/LibMultiTokenStakingStorage.sol	5e276b3caeddd39278b2d3d12da660235730a6ec
contracts/staking/MultiTokenStakingInit.sol	24fddfb4b448191e2d5f023b2f0187c4d53c9cbc
contracts/staking/facets/StackQueryFacet.sol	575fb45d14905394bb621022f68519329c9ca2dd
contracts/staking/facets/StackOperationFacet.sol	2006a04c099d81c6d93f328652f93592fdab92d4

SALUS