verichains

SECURITY AUDIT OF

LIQUIDITY VAULT SMART CONTRACTS

Public Report

Jun 10, 2025

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jun 10, 2025. We would like to thank the Krystal for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Liquidity Vault Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team identified some vulnerabilities in the smart contracts code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Liquidity Vault Smart Contracts

Krystal is a Decentralized, Multichain Liquidity Farming Agent that enables users to earn yield by providing liquidity across various DEXs and blockchains.

Krystal's Liquidity Vault is a powerful new tool that lets users manage, automate, and optimize liquidity provisioning across DeFi protocols through a Vault Interface.

It's a dynamic evolution from static liquidity strategies, giving users more flexibility, automation, and performance opportunities.

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the Liquidity Vault Smart Contracts. It was conducted on commit `0a5d5679700bdf2bb33f9a3b7a511bb15be6c0a6` from git repository link: *https://github.com/KrystalDeFi/krystal-vault-contracts-v2/*.

The latest version of the following files were made available in the course of the review:

SHA256 Sum	File
687c714197ee8a8dc686ef8e6ec260da22b3ebde3021790b2a92976f2107cc17	contracts/core/ConfigManager.sol
3f2dca4467fd0645266f20f4be8e6e00901d4d6d2f035d7930b74b11a8133b46	contracts/core/PoolOptimalSwapper.sol
26cd9071080d9d34ec8b0fed2ef15c9608c3794d55ca139e2db5a5bbfd199c15	contracts/core/Vault.sol
05fb2459f785591b8dfcf994d157f4ad9024f766f1dde6c1f87a5b4338e064cf	contracts/core/VaultFactory.sol
53a39e05d7c9a459df479656a148c61eee84fd0b2df7a0e0355b8578ea188da7	contracts/interfaces/ICommon.sol
59ad9f85be2a4ebf00637c515ee539d5111e6c844ff222c89404c7332f2cf526	contracts/interfaces/ICreateX.sol
94959413f6d287fa16d0219aba3aad5ac00e103099471079d3031d3fd51aa251	contracts/interfaces/IWETH9.sol
2ada4ed52baf5ba3ef36985960142413b90d933b11b8d20b24415aa2a7373f7e	contracts/libraries/strategies/LpUniV3StructHash.sol

1dd45a00de8c4a3ebfb00ce345f3bad91be547be7da43c7408da9de5e9485798	contracts/libraries/AssetLib.sol
55502bfa16b793d2ed552ff69cea3a287b151a35153665277efa6458b8a2573c	contracts/libraries/InventoryLib.sol
eabe32cc549fa3e41888601aa80e41521ebc335d96c160b527cde4a2f317d486	contracts/libraries/OptimalSwap.sol
044592aae336db73c62a8950c771d600f6246e7b7182de13abcd73fa1f34f2a1	contracts/strategies/lpUniV3/CustomEIP712.sol
e4157489d6f6e37b58e72d052eaca087385003c24c750bce712fba77e68fc7e7	contracts/strategies/lpUniV3/LpFeeTaker.sol
9dc2791febde6425753a2930a2126d25a5b51e88d453c5be594e995fc4688814	contracts/strategies/lpUniV3/LpStrategy.sol
2144d327a00240edea25ddbc0a6ee8c2ade078ed463aaada922134e3eea55a68	contracts/strategies/lpUniV3/LpValidator.sol
bf02ffb64805232b2d66ef1d8e3d50dded282049afe47fe340c9708b14cc3b13	contracts/strategies/lpUniV3/VaultAutomator.sol
81a08d497180368d94ccab9bf1fb2009330d502cfc09f6f4887bcaa4c785ca10	contracts/strategies/merkl/MerklAutomator.sol
0e36e2f8ccb9c2943303fa9866f0db8c7c177255504b31bdfc50eddfd58a8eeb	contracts/strategies/merkl/MerklStrategy.sol

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence

- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Krystal acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Krystal understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Krystal agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Krystal will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Krystal, the final report will be considered fully accepted by the Krystal without the signature.

2. AUDIT RESULT

2.1. Overview

The Liquidity Vault Smart Contracts was written in `Solidity` language, with the required version to be `^0.8.28`.

2.1.1. Vault contract

This contract represents an individual investment vault where users can deposit and withdraw a specific principal token. It manages these assets by allocating them to various approved investment strategies. The vault itself functions as an `ERC20` token, with shares representing a user's stake. It incorporates role-based access control for administrative functions and relies on the `ConfigManager` for operational settings, permissions, and fee structures. Key operations like deposits and withdrawals can be paused, and authorized "automators" can sweep mistakenly sent tokens. When the `allowDeposit` flag is set to `false`, the vault will not allow any deposits and withdrawals will be disabled. When it is set to `true`, the vault will only support deposits when the pools specified by `LpStrategy` are stable and have the slippage not greater than `configManager.maxHarvestSlippage`.

2.1.2. VaultFactory contract

This contract serves as a factory to create new Vault instances. It employs a deterministic cloning mechanism to deploy new vaults based on a pre-set implementation address. The factory handles the initial deposit of principal tokens (ETH or ERC20) into these newly created vaults. It maintains a record of all created vaults, both globally and per creator address. The contract is ownable, allowing the owner to manage settings like the vault implementation and `ConfigManager` address, and pausable, enabling the temporary suspension of new vault creation.

2.1.3. LpStrategy contract

This contract implements a strategy for managing liquidity positions, primarily on `Uniswap V3` and compatible automated market makers (AMMs). It interacts with the Nonfungible Position Manager (NFPM) to handle LP positions, which are represented as NFTs. The strategy can calculate the value of these LP positions in terms of a principal token. Its core convert function acts as a router, enabling various complex LP management operations such as minting new positions (potentially after swapping input tokens), increasing or decreasing liquidity in existing positions (again, with optional swaps), rebalancing positions by adjusting their price ranges, and compounding earned fees back into the positions. It relies on an `IOptimalSwapper` for efficient token exchanges, an `ILpValidator` for validating operations, and an `ILpFeeTaker` for processing collected fees.

2.1.4. ConfigManager contract

This contract acts as a central hub for all configuration settings across the protocol. It manages whitelists for approved strategies, swap routers, automator addresses (which can perform privileged actions), and signers. It stores strategy-specific configurations, defines categories for "typed tokens" (like stablecoins), and sets global parameters such as the maximum number of positions a vault can hold, maximum harvest slippage, and a global pause flag for vault operations. Additionally, it holds the fee structures for both public (deposit-allowed) and private vaults, and is ownable, allowing an administrator to update all these settings.

2.1.5. PoolOptimalSwapper contract

This contract is designed to execute token swaps in the most efficient way, primarily by interacting with `Uniswap V3` (and compatible) liquidity pools. It includes the necessary callback functions required by these decentralized exchanges to finalize swaps. Its core functionality involves calculating the optimal swap parameters and executing the trade, aiming to minimize slippage and maximize returns. It also provides a simpler function for direct exact-input swaps.

2.1.6. LpValidator contract

The `LpValidator` contract is designed to enforce rules and constraints for operations within the `LpStrategy`. It's used to validate parameters like tick ranges, pool configurations, and price sanity before the `LpStrategy` executes actions like minting new positions or rebalancing existing ones. For instance, `LpStrategy` calls `validator.validateConfig(...)` or `validator.validateTickWidth(...)` to ensure the proposed LP parameters are acceptable according to the `vaultConfig`. The `uint32 secondLastTimestamp` variable in `LpValidator` likely plays a role in time-sensitive validations, possibly related to oracle price checks or TWAP calculations to ensure price data is fresh and not manipulated.

2.1.7. VaultAutomator contract

The `VaultAutomator` contract enables off-chain services or authorized operators to execute specific, pre-approved functions on a vault, such as allocate (which can trigger rebalancing or compounding in `LpStrategy`), or sweeping tokens. It achieves this by requiring a signed order from the vault owner, which `VaultAutomator` verifies before execution. This allows for automated management of vault positions.

2.2. Findings

During the audit process, the audit team identified some issues in the contract code.

#	Title	Severity	Status
1	Incorrect Calculation of TotalValue in `Vault`	HIGH	Acknowledged
2	Incorrect Calculation of `denominator` in deposit function	HIGH	Fixed
3	Incorrect parameter when initialize a vault	HIGH	Fixed
4	Unauthenticated and Replayable Signatures in `executeAllocate` function	MEDIUM	Acknowledged
5	Incorrect Array Index in Function Documentation	INFO	Acknowledged
6	Using `calldata` instead of `memory` for function parameter to save gas	INFO	Acknowledged
7	Out of bound access array when the length of `_typedTokenTypes` is less than `_typedTokens` length	INFO	Acknowledged

2.2.1. Incorrect Calculation of TotalValue in `Vault` HIGH

- Affected files: contracts/core/Vault.sol

During the `deposit` process, the vault contract should add up every asset it holds—including those staked in liquidity pools—to decide how many shares to mint for the user. Unfortunately, the `getTotalValue` function ignores ERC-20 tokens left in the vault's inventory (extra tokens that accumulate there after swaps and allocations). Because these "spare" assets aren't counted, the reported `totalValue` is lower than the vault's true holdings, leading the contract to mint more shares to the user than intended.

```
function getTotalValue() public view returns (uint256 totalValue) {
  uint256 length = inventory.assets.length;

  AssetLib.Asset memory currentAsset;

  for (uint256 i; i < length;) {
    currentAsset = inventory.assets[i];
    if (currentAsset.strategy != address(0) && currentAsset.amount != 0) {
      totalValue += IStrategy(currentAsset.strategy).valueOf(currentAsset,
vaultConfig.principalToken); //<-- calculate all assets in deposited pool
    } else if (currentAsset.token == vaultConfig.principalToken) {
      totalValue += currentAsset.amount; <-- only calculate the pricipal assets
    }
      //<-- Missing another ERC20 assets
    unchecked {
      i++;
    }
  }
}

  function deposit(uint256 principalAmount, uint256 minShares)
  external
  payable
  nonReentrant
  whenNotPaused
  returns (uint256 shares)
{
────────────────────────────────────────────────────────────────────────

  uint256 totalSupply = totalSupply();
  // update total value after distributing the principal amount to the strategies
  totalValue = getTotalValue() - principalAmount; //<-- The getTotalValue is less than the
real assets because of missing the ERC20 assets(the spare assets were added to inventory
during swap and allocate) in inventory

  uint256 totalValueAfterDepositAndSubtractInputPrincipalAmount = totalValue;
  shares =
    totalSupply == 0 ? principalAmount * SHARES_PRECISION :
FullMath.mulDiv(principalAmount, totalSupply, totalValue);

  require(shares >= minShares, InsufficientShares());
  _mint(_msgSender(), shares);

  emit TotalValueUpdated(totalValueBeforeDeposit,
totalValueAfterDepositAndSubtractInputPrincipalAmount);
  emit VaultDeposit(vaultFactory, _msgSender(), principalAmount, shares);

  return shares;
}
```

RECOMMENDATION

- The `getTotalValue` must cover all ERC20 assets value.

UPDATES

When `allowDeposit` is set to false (so only the admin can allocate inventory), adding liquidity from the vault to a low-liquidity pool with the single-asset method will incur high slippage. This leaves a sizable amount of spare tokens—far more than “dust.” The team should take this scenario into account.

UPDATES

- **Jun 10, 2025**: This issue has been acknowledged by Krystal team.

2.2.2. Incorrect Calculation of `denominator` in deposit function HIGH

- Affected files: contracts/core/Vault.sol

Inside the `deposit function`, `totalValue` serves as the denominator when determining how many shares to mint for the user. In the current implementation, the contract first receives an amount of the principal token, then swaps it for various ERC-20 assets and supplies them as liquidity. Because each swap and liquidity addition incurs fees and slippage, the actual value deposited into the pools is lower than the original principal-token amount. Consequently, `getTotalValue` does not rise by the full amount of the deposit, so its reported figure is understated. Using this understated `totalValue` in the share-minting make the user gain more share than the expected value.

```
function deposit(uint256 principalAmount, uint256 minShares)
    external
    payable
    nonReentrant
    whenNotPaused
    returns (uint256 shares)
  {
    require(_msgSender() == vaultOwner || vaultConfig.allowDeposit, DepositNotAllowed());
    require(principalAmount != 0, InvalidAssetAmount());

    uint256 length = inventory.assets.length;

    for (uint256 i; i < length;) {
      AssetLib.Asset memory currentAsset = inventory.assets[i];
      if (currentAsset.strategy != address(0) && currentAsset.amount != 0)
_harvest(currentAsset, 0);

      unchecked {
        i++;
      }
    }
    address principalToken = vaultConfig.principalToken;
    uint256 totalValue = getTotalValue(); //<-- The first totalValue before depositing

    if (msg.value > 0) {
      require(principalToken == WETH, InvalidAssetToken());
      require(principalAmount == msg.value, InvalidAssetAmount());
      IWETH9(principalToken).deposit{ value: msg.value }();
    } else {
      IERC20(principalToken).safeTransferFrom(_msgSender(), address(this),
principalAmount);
    }
    inventory.addAsset(AssetLib.Asset(AssetLib.AssetType.ERC20, address(0), principalToken,
0, principalAmount));

    for (uint256 i; i < inventory.assets.length;) {
      AssetLib.Asset memory currentAsset = inventory.assets[i];

      if (currentAsset.strategy != address(0) && currentAsset.amount != 0) {
        uint256 strategyPosValue = IStrategy(currentAsset.strategy).valueOf(currentAsset,
principalToken);
        if (strategyPosValue != 0) {
          uint256 pAmountForStrategy = FullMath.mulDiv(principalAmount, strategyPosValue,
totalValue);
          inventory.removeAsset(currentAsset);
          inventory.removeAsset(
            AssetLib.Asset(AssetLib.AssetType.ERC20, address(0), principalToken, 0,
pAmountForStrategy)
          );
          bytes memory cData = abi.encodeWithSelector(
            IStrategy.convertFromPrincipal.selector, currentAsset, pAmountForStrategy,
vaultConfig
          );
          bytes memory returnData = _delegateCallToStrategy(currentAsset.strategy, cData);
//<-- convert deposited principal token to other ERC20 assets and then addLiquidity to the
pools
          _addAssets(abi.decode(returnData, (AssetLib.Asset[])));
        }
      }

      unchecked {
        i++;
      }
    }

    uint256 totalSupply = totalSupply();
    // update total value after distributing the principal amount to the strategies
    totalValue = getTotalValue() - principalAmount; //<-- The `getTotalValue` is increased
but the real increasing value is less than the principalAmount (because of the fee and
slippage). So the totalValue in this step is less than the totalValueBeforeDeposit

    shares =
      totalSupply == 0 ? principalAmount * SHARES_PRECISION :
FullMath.mulDiv(principalAmount, totalSupply, totalValue); //<-- The `shares` is calculated
based on the wrong `totalValue` make the user gain more share than the expected value.

    require(shares >= minShares, InsufficientShares());
    _mint(_msgSender(), shares);

    emit TotalValueUpdated(totalValueBeforeDeposit,
totalValueAfterDepositAndSubtractInputPrincipalAmount);
    emit VaultDeposit(vaultFactory, _msgSender(), principalAmount, shares);
    return shares;
```

RECOMMENDATION

- Team should review this logic carefully. To ensure user not gain more share than the expected value, we should use the max value between `totalValueBeforeDeposit` and `totalValueAfterAddliquidity` as `denominator` in the `shares` calculation..

UPDATES

- **Jun 10, 2025**: This issue has been acknowledged and fixed by Krystal team.

2.2.3. Incorrect parameter when initialize a vault HIGH

- Affected files: contracts/core/VaultFactory.sol

When `createVault()` in VaultFactory contract, when the principal token is WETH, the `principalAmount` is overwritten by `msg.value`. But in step `IVault(vault).initialize()`, it still uses the original value from `params.principalTokenAmount`. This mismatch can lead to incorrect initialization of the vault, causing the inventory to record the wrong value and the vault to mint an incorrect amount of vault tokens.

```
function createVault(VaultCreateParams memory params) external payable override
whenNotPaused returns (address vault) {
    // ...

  address principalToken = params.config.principalToken;
  uint256 principalAmount = params.principalTokenAmount;

  if (msg.value > 0) {
    require(principalToken == WETH, InvalidPrincipalToken());
    principalAmount = msg.value; // <-- OVERWRITE!!!
    IWETH9(WETH).deposit{ value: msg.value }();
    IERC20(WETH).safeTransfer(vault, msg.value);
  } else if (principalAmount > 0) {
    IERC20(principalToken).safeTransferFrom(sender, vault, principalAmount);
  }

  IVault(vault).initialize(params, sender, configManager, WETH); // <-- USES old
params.principalTokenAmount

  // ...
}

// In contracts/core/Vault.sol
function initialize(VaultCreateParams calldata params, address _owner, address
_configManager, address _weth)
```

```
  public
  initializer
{
    // ...

  uint256 principalAmount = params.principalTokenAmount;
  address principalToken = params.config.principalToken;

  // Initialize first asset in inventory
  inventory.addAsset(AssetLib.Asset(AssetLib.AssetType.ERC20, address(0), principalToken,
0, principalAmount));

  // Mint shares only if principalAmount > 0 to save gas on unnecessary operations
  if (principalAmount > 0) {
    unchecked {
      uint256 mintAmount = principalAmount * SHARES_PRECISION;
      _mint(_owner, mintAmount);
      emit VaultDeposit(_msgSender(), _owner, principalAmount, mintAmount);
    }
  }
}
```

RECOMMENDATION

Update the params passed to `IVault(vault).initialize()` to use the correct `principalAmount` that was set after the WETH deposit.

UPDATES

- **Jun 10, 2025**: This issue has been acknowledged and fixed by Krystal team.

2.2.4. Unauthenticated and Replayable Signatures in `executeAllocate` function MEDIUM

- Affected file: contracts/strategies/merkl/MerklStrategy.sol

The executeAllocate function in the MerklStrategy contract is meant to execute a signed instruction, but it suffers from two flaws:`

- Missing Authentication and Parameter Binding: the function does not authenticate the caller or bind the signature to all of the instruction's parameters. As a result, any address can invoke executeAllocate, tweak certain fields, and redirect the call to a different Vault or Strategy contract.
- Replayable Signatures: the data bundled for signature verification includes only a timestamped deadline. Until that deadline is reached, the same signature can be reused indefinitely,

```
function executeAllocate(
    IVault vault, //<-- any addresses can snip admin tx and resend with the same paramater
(with old sig) to trigger new vault/strategy
    AssetLib.Asset[] memory inputAssets,
    IStrategy strategy,
    uint64 gasFeeX64,
    bytes calldata allocateData,
    bytes calldata,
    bytes calldata
  ) external whenNotPaused {
    require(inputAssets.length == 0, InvalidAssetStrategy());
    Instruction memory instruction = abi.decode(allocateData, (Instruction));
    require(instruction.instructionType ==
uint8(IMerklStrategy.InstructionType.ClaimAndSwap), InvalidInstructionType());
    IMerklStrategy.ClaimAndSwapParams memory claimParams =
      abi.decode(instruction.params, (IMerklStrategy.ClaimAndSwapParams));

    // Verify the signer is whitelisted
    bytes32 messageHash = keccak256(
      abi.encodePacked(
        claimParams.distributor,
        claimParams.token,
        claimParams.amount,
        claimParams.proof,
        claimParams.swapRouter,
        claimParams.swapData,
        claimParams.amountOutMin,
        claimParams.deadline
      )
    );
    address signer = ECDSA.recover(messageHash, claimParams.signature);
    require(configManager.isWhitelistSigner(signer), InvalidSigner());
    require(block.timestamp <= claimParams.deadline, SignatureExpired());

    vault.allocate(inputAssets, strategy, gasFeeX64, allocateData);
  }
```

RECOMMENDATION

The message hash must incorporate both the vault address and the strategy address so that a signature cannot be replayed against another target.

Moreover, to stop a signature from being reused before its deadline, introduce an ever-increasing nonce. By requiring each transaction to carry a unique nonce, the same signature can be accepted only once.

UPDATES

- **Jun 10, 2025**: This issue has been acknowledged by Krystal team.

2.2.5. Incorrect Array Index in Function Documentation INFORMATIVE

Position

- contracts/strategies/lpUniV3/LpStrategy.sol#L430
- contracts/strategies/lpUniV3/LpStrategy.sol#L613

Description

In the `swapAndIncreaseLiquidity` function documentation at line 428 to 433, the comment incorrectly states that `assets[2] = lpAsset` (implying the asset array length must be at least 3), but the actual implementation shows that the function only accepts 2 assets (enforced by `require(assets.length == 2, InvalidNumberOfAssets())` at line 439) and the lpAsset is accessed at index 1 (`AssetLib.Asset memory lpAsset = assets[1];` at line 443).

```
/// @notice Swaps the principal token to the other token and increases the liquidity of the
position
  /// @param assets The assets to swap and increase liquidity, assets[2] = lpAsset
  /// @param params The parameters for swapping and increasing the liquidity
  /// @param vaultConfig The vault configuration
  /// @return returnAssets The assets that were returned to the msg.sender
  function swapAndIncreaseLiquidity(
    AssetLib.Asset[] calldata assets,
    SwapAndIncreaseLiquidityParams memory params,
    VaultConfig calldata vaultConfig
  ) internal returns (AssetLib.Asset[] memory returnAssets) {
    require(assets.length == 2, InvalidNumberOfAssets());
    require(assets[0].token == vaultConfig.principalToken, InvalidAsset());

    AssetLib.Asset memory asset0 = assets[0];
    AssetLib.Asset memory lpAsset = assets[1];

    // ...
```

In the `swapAndRebalancePosition` function documentation at line 613, the comment incorrectly states that the function takes `assets[0] = principalToken, assets[1] = lpAsset`, but the actual implementation shows that the function only accepts 1 asset (enforced by `require(assets.length == 1, InvalidNumberOfAssets())` at line 624) and only uses `assets[0]` which is the lpAsset, not the principalToken.

```
/// @notice Swaps the principal token to the other token and rebalances the position
 /// @param assets The assets to swap and rebalance, assets[0] = principalToken, assets[1]
= lpAsset
 /// @param params The parameters for swapping and rebalancing the position
 /// @param vaultConfig The vault configuration
 /// @param feeConfig The fee configuration
 /// @return returnAssets The assets that were returned to the msg.sender
 function swapAndRebalancePosition(
   AssetLib.Asset[] calldata assets,
   SwapAndRebalancePositionParams memory params,
   VaultConfig calldata vaultConfig,
   FeeConfig calldata feeConfig
 ) internal returns (AssetLib.Asset[] memory returnAssets) {
   require(assets.length == 1, InvalidNumberOfAssets());

   // ...
```

RECOMMENDATION

Update the documentation to reflect the actual implementation.

UPDATES

- **Jun 10, 2025**: This issue has been acknowledged by Krystal team.

2.2.6. Using `calldata` instead of `memory` for function parameter to save gas INFORMATIVE

- Affected files: contracts/core/ConfigManager.sol,...

Using memory for function parameters can lead to unnecessary gas costs, especially when the data is not modified within the function. Using `calldata` can save gas by avoiding the need to copy data into memory. Like the `` `initialize `` function in ConfigManager.sol, it can be changed to use `calldata` for the `params` parameter to save gas.

```
function initialize(
  address _owner,
  address[] memory _whitelistStrategies,
  address[] memory _whitelistSwapRouters,
  address[] memory _whitelistAutomator,
  address[] memory _whitelistSigners,
  address[] memory _typedTokens,
  uint256[] memory _typedTokenTypes,
  uint16 _vaultOwnerFeeBasisPoint,
  uint16 _platformFeeBasisPoint,
  uint16 _privatePlatformFeeBasisPoint,
  address _feeCollector,
  address[] memory _strategies,
  address[] memory _principalTokens,
  bytes[] memory _configs
) public initializer {
  // ...
}
```

RECOMMENDATION

Team should review the codebase and identify other instances where `memory` can be replaced with `calldata` for function parameters that are not modified. This can lead to significant gas savings, especially in frequently called functions.

UPDATES

- **Jun 10, 2025**: This issue has been acknowledged by Krystal team.

2.2.7. Out of bound access array when the length of `_typedTokenTypes` is less than `_typedTokens` length INFORMATIVE

- Affected files: contracts/core/ConfigManager.sol

When initializing the `ConfigManager`, there is a potential out-of-bounds access if the length of `_typedTokenTypes` does not match the length of `_typedTokens`. This can lead to unexpected behavior or even a revert if the code tries to access an index that does not exist.

```
function initialize(
   address _owner,
   address[] memory _whitelistStrategies,
   address[] memory _whitelistSwapRouters,
   address[] memory _whitelistAutomator,
   address[] memory _whitelistSigners,
   address[] memory _typedTokens,
   uint256[] memory _typedTokenTypes,
   uint16 _vaultOwnerFeeBasisPoint,
   uint16 _platformFeeBasisPoint,
   uint16 _privatePlatformFeeBasisPoint,
   address _feeCollector,
   address[] memory _strategies,
   address[] memory _principalTokens,
   bytes[] memory _configs
 ) public initializer {
   __Ownable_init(_owner);

   uint256 length = _typedTokens.length;

   for (uint256 i; i < length;) {
     typedTokens.set(_typedTokens[i], _typedTokenTypes[i]); //<-- Out of bound access when
`_typedTokenTypes` is shorter than `_typedTokens`

     unchecked {
       i++;
     }
   }
```

RECOMMENDATION

Add a check to ensure that the lengths of `_typedTokens` and `_typedTokenTypes` are equal before proceeding with the initialization. If they are not equal, revert the transaction with an appropriate error message.

UPDATES

- **Jun 10, 2025**: This issue has been acknowledged by Krystal team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jun 10, 2025*	Public Report	Verichains Lab

Table 2. Report versions history