



WOWMAX SMART CONTRACT CODE REVIEW & SECURITY ASSESSMENT

REPORT

08.08.2025

Table of content

Pages

03-05

Project Summary	03
Project Scope	03
Project Overview	03
Findings Summary	04
Audit team & Contributions	05

05- 26

Findings	05
High	05
Low	21
Informational	23

27

Disclaimer	27
-------------------	----

28

About Hakira	28
---------------------	----

Project Summary

Project Scope

Project Name	Repository	Reviewed Commits	Platform
Wowmax Copy Trading	https://github.com/wowswapio/wowmax-copy-trading	ceea2f3 (initial commit) 002cc8d (latest commit)	EVM/Solidity

Project Overview

The project "WoW Max" is a decentralized exchange (DEX) aggregator designed with the objective of optimizing exchange amounts. Its primary function involves analyzing token prices across multiple DEX platforms and identifying the most efficient exchange path among various exchange protocols.

This document describes the issues and weaknesses of WoW Max Contracts using the manual code review findings. The work was undertaken from Aug 1, 2025 to Aug 8, 2025.

The following contract list is included in our scope:

- contracts/PricesVerifier.sol
- contracts/WowmaxCopyTradingFundsManager.sol
- contracts/WowmaxCopyTradingVault.sol

Findings Summary

Severity	Finding	Confirmed	Fixed
Critical	-	-	-
High	3	3	3
Medium	-	-	-
Low	2	2	1
Informational	4	4	3
Total	9	9	7

Audit Team & Contributions

Lead Security Researcher:

Tigran Piliposian - Led the audit engagement, including technical scoping, manual code review, and risk classification. Played a key role in identifying and validating critical and high-severity vulnerabilities, ensuring the accuracy and depth of findings throughout the assessment lifecycle.

Contributors:

The Hakira security team assisted with code review, threat modeling, and exploit simulation to ensure comprehensive coverage of the audited scope.

Findings

This section contains the list of discovered findings.

H-01. Fee mismatch in share calculation allows value extraction from early depositors

Status: Fixed

Severity: High

Path: [contracts/WowmaxCopyTradingVault.sol#L161-L168](#)

Description:

The `makeDeposit()` function in `WowmaxCopyTradingVault.sol` calculates shares based on gross deposit amounts (before fees) but withdrawals are based on net amounts (after fees). This mismatch creates an unfair value transfer mechanism where users who deposit during high-fee periods gain value at the expense of users who deposited during low-fee periods.

```
function makeDeposit(Deposit calldata deposit) external onlyFundsManager nonReentrant whenNotPaused override {
    require(deposit.leader != address(0), "WOWMAX: Invalid leader address");
    require(verifyPrices(deposit.priceData, priceSigner), "WOWMAX: Invalid prices signature");
    if (leaders.add(deposit.leader)) {
        emit LeaderAdded(deposit.leader);
    }
    followerLeaders[deposit.to].add(deposit.leader);
    leaderFollowers[deposit.leader].add(deposit.to);
    uint256 followersTVL = getFollowersUSDValue(deposit.leader, deposit.priceData.prices);
    for (uint256 i = 0; i < deposit.amounts.length; i++) {
        require(deposit.amounts[i].value > 0, "WOWMAX: Amount must be greater than 0");
        IERC20(deposit.amounts[i].token).safeTransferFrom(msg.sender, address(this), deposit.amounts[i].value);
        increaseBalance(deposit.leader, deposit.amounts[i].token, deposit.amounts[i].value);
        emit FollowerDeposit(deposit.leader, deposit.to, leaderNonces[deposit.leader], deposit.amounts[i].token, deposit.amounts[i].value);
    }
    @> uint256 depositUSDValue = getDepositUSDValue(deposit);
    uint256 totalShare = totalShares[deposit.leader];
    uint256 share = totalShare == 0 ? depositUSDValue : depositUSDValue * totalShare / followersTVL;
    shares[deposit.leader][deposit.to] += share;
    totalShares[deposit.leader] += share;
    if (followerDepositNonces[deposit.to][deposit.leader] == 0) {
```

```

        followerDepositNonces[deposit.to][deposit.leader] = leaderNonces[depo
sit.leader];
    }

    emit Invest(deposit.leader, deposit.to, deposit.investmentToken, deposit.inv
estmentAmount, deposit.referralCode);
    emit FollowerShareUpdate(deposit.leader, deposit.to, shares[deposit.leader]
[deposit.to]);
    emit LeaderSharesUpdate(deposit.leader, totalShares[deposit.leader]);
}

```

Shares are calculated from gross amounts (before fees) while portfolio balances reflect net amounts (after fees). This creates a mathematical inconsistency that enables unfair value extraction, as then when withdrawing it takes the actual balances and the shares weight:

```

function withdraw(address leader) external whenNotPaused nonReentrant {
    address follower = msg.sender;
    require(shares[leader][follower] > 0, "WOWMAX: No shares to withdraw");
    uint256 share = shares[leader][follower];
    shares[leader][follower] = 0;
    uint256 totalShare = totalShares[leader];
    totalShares[leader] -= share;
    bool cleanUpTokens;
    for (uint256 i = 0; i < leaderTokens[leader].length(); i++) {
        address token = leaderTokens[leader].at(i);
        @>     uint256 amount = balances[leader][token] * share / totalShare;
        balances[leader][token] -= amount;
        if (balances[leader][token] == 0) {
            cleanUpTokens = true;
        }
        IERC20(token).safeTransfer(follower, amount);
        emit FollowerWithdraw(leader, follower, leaderNonces[leader], token, am
ount);
    }
}

```

```

followerLeaders[follower].remove(leader);
leaderFollowers[leader].remove(follower);
if (cleanUpTokens) {
    cleanUpLeaderTokens(leader);
}
followerDepositNonces[follower][leader] = 0;
emit FollowerShareUpdate(leader, msg.sender, 0);
emit LeaderSharesUpdate(leader, totalShares[leader]);
}

```

Example Scenario:

Alice deposits `1000 USDT` when `fee = 5%`. Then fee increases from `5` to `10`. Bob deposits `1000 USDT` when `fee = 10%`.

1. Alice deposits:

- Gross: `1000 USDT`
- Fee: `50 USDT (5%)`
- Net to portfolio: `950 USDT`
- Shares received: `1000e18` (first deposit, `shares = depositUSDValue`)

1. Bob deposits:

- Gross: `1000 USDT`
- Fee: `100 USDT (10%)`
- Net to portfolio: `900 USDT`
- Current `followersTVL = 950` (Alice's net contribution)
- Shares received: `1000 * 1000e18 / 950 = 1052.63e18`

Portfolio State:

Total shares: `1000e18 + 1052.63e18 = 2052.63e18`

Total actual value: `950 + 900 = 1850 USDT`

If they want to withdraw:

- Alice receives: $1850 * 1000\text{e}18 / 2052.63\text{e}18 = 901.3 \text{ USDT}$
- Bob receives: $1850 * 1052.63\text{e}18 / 2052.63\text{e}18 = 948.7 \text{ USDT}$

so

Alice contributed 950 USDT net, received 901.3 USDT . Bob contributed 900 USDT net, received 948.7 USDT .

After Bob performs multiple deposit and withdrawal cycles, he extracts significant value from Alice.

PoC (Proof of Concept):

The issue is demonstrated by the following test:

```
it("Should demonstrate unfair share allocation due to gross vs net calculation", () => {
  sync () => {
    // Set initial fee to 5%
    await vault.setFee(500); // 5% fee

    // Alice deposits 1 WBTC with 5% fee (worth $100,000)
    await makeDeposit(follower, leader, [[wbtc, 1]]);

    // Alice's shares and net deposit
    const aliceShares = await vault.getShare(leader, follower);
    const aliceNet = ethers.parseUnits("0.95", 8);
    console.log("Alice shares (gross):", ethers.formatUnits(aliceShares.amounts[0], 18));
    console.log("Alice net deposit (after fee):", ethers.formatUnits(aliceNet, 8));

    // Change fee to 10%
    await vault.setFee(1000); // 10% fee

    // Bob deposits 1 WBTC with 10% fee (worth $100,000)
    await makeDeposit(follower2, leader, [[wbtc, 1]]);

    // Bob's shares and net deposit
    const bobShares = await vault.getShare(leader, follower2);
    const bobNet = ethers.parseUnits("0.9", 8);
```

```

        console.log("Bob shares (gross):", ethers.formatUnits(bobShares.amounts[0], 1
8));
        console.log("Bob net deposit (after fee):", ethers.formatUnits(bobNet, 8));

        // Check total portfolio state
        const totalBalance = await vault.getFollowersBalances(leader);
        console.log("Total WBTC in vault:", ethers.formatUnits(totalBalance.amounts
[0], 8));

        // Alice withdraws
        const aliceBalanceBefore = await wbtc.balanceOf(follower);
        await vault.connect(follower).withdraw(leader);
        const aliceBalanceAfter = await wbtc.balanceOf(follower);
        const aliceReceived = aliceBalanceAfter - aliceBalanceBefore;
        console.log("Alice received on withdraw:", ethers.formatUnits(aliceReceived,
8));
        console.log("Alice loss:", ethers.formatUnits(aliceNet - aliceReceived, 8));

        // Bob withdraws
        const bobBalanceBefore = await wbtc.balanceOf(follower2);
        await vault.connect(follower2).withdraw(leader);
        const bobBalanceAfter = await wbtc.balanceOf(follower2);
        const bobReceived = bobBalanceAfter - bobBalanceBefore;
        console.log("Bob received on withdraw:", ethers.formatUnits(bobReceived, 8));
        console.log("Bob gain:", ethers.formatUnits(bobReceived - bobNet, 8));

        // The difference should be equal (Alice loses what Bob gains)
        const aliceLoss = aliceNet - aliceReceived;
        const bobGain = bobReceived - bobNet;
        expect(aliceLoss).to.be.equal(bobGain);
    });

```

Output:

Alice shares (gross): 10000000000000000000000000
Alice net deposit (after fee): 0.95
Bob shares (gross): 10000000000000000000000000
Bob net deposit (after fee): 0.9
Total WBTC in vault: 1.85
Alice received on withdraw: 0.90128205
Alice loss: 0.04871795
Bob received on withdraw: 0.94871795
Bob gain: 0.04871795

Impact:

The economic impact of this vulnerability is substantial. Users who deposit during lower fee periods risk losing significant value to users who deposit during higher fee periods, even when no malicious intent exists. This creates an unfair system where the timing of deposits relative to fee changes determines user outcomes rather than the actual value contributed.

The vulnerability also creates perverse incentives for user behavior. Users may be incentivized to wait for higher fee periods to extract value from earlier depositors, while early depositors face the risk of losing value to later users. This undermines trust in the protocol and creates uncertainty about actual returns.

Importantly, this vulnerability is systemic and affects all users regardless of their intentions. Bob in our example doesn't need to be malicious to benefit from the flaw; the mathematical inconsistency automatically transfers value from Alice to Bob based on their different fee rates.

Recommendation:

Modify the `makeDeposit` function calculations so when it calculates the shares it is taking the net values rather than gross values.

H-02. Same-token swaps cause permanent loss and drainage of stuck tokens

Status: Fixed

Severity: High

Path: [contracts/WowmaxCopyTradingFundsManager.sol#L207-L237](#)

Description:

The `WowmaxCopyTradingFundsManager.sol` contract contains a vulnerability in the `swapManyToOne()` function. The external `swapManyToOne()` function lacks proper balance tracking to handle cases where input and output tokens are the same.

```
function swapManyToOne(MultiSwap calldata multiSwap, address to) external  
nonReentrant override returns (uint256 amountOut) {  
    require(to != address(0), "Funds Manager: invalid recipient");  
    amountOut = _swapManyToOne(multiSwap);  
    IERC20(multiSwap.token).safeTransfer(to, amountOut);  
}
```

If a user transfers tokens to the contract and attempts to swap a token to itself (e.g., USDT→USDT), the funds become stuck due to a `continue` statement in the `_swapManyToOne()` function. Any other user can then come and swap those stuck tokens to another token (e.g., USDT→WBTC), effectively stealing the original user's funds.

```
function _swapManyToOne(MultiSwap calldata multiSwap) private returns (uint256 amountOut) {  
    require(allowedRouters.contains(multiSwap.router), "Funds Manager: router not allowed");  
    require(multiSwap.deadline >= block.timestamp, "Funds Manager: expire d");  
    uint256 amountIn;
```

```

        uint256 swapAmountOut;
        address tokenIn;
        uint256 balanceBefore = IERC20(multiSwap.token).balanceOf(address(this));
    s));
        uint256 balanceAfter;
        uint256 len = multiSwap.swaps.length;
        for (uint256 i = 0; i < len; i++) {
            amountIn = multiSwap.swaps[i].amount;
            tokenIn = multiSwap.swaps[i].token;
            if (tokenIn == multiSwap.token) {
                continue;
            }
            IERC20(tokenIn).safeIncreaseAllowance(multiSwap.router, amountIn);
            (bool success,) = multiSwap.router.call(multiSwap.swaps[i].swapData);
            require(success, "Funds Manager: swap failed");
            balanceAfter = IERC20(multiSwap.token).balanceOf(address(this));
            swapAmountOut = balanceAfter - balanceBefore;
            require(swapAmountOut >= multiSwap.swaps[i].minAmountOut, "Funds
Manager: insufficient output amount");
            balanceBefore = balanceAfter;
            amountOut += swapAmountOut;
        }
        return amountOut;
    }

```

The `swapManyToOne()` is an external function, and it works with swapping the funds that user has already transferred beforehand, so if there is any funds stuck in the contract anyone can drain that funds using this function. Now the internal `_swapManyToOne()` function has an `if` - `continue`, when a user attempts to swap a token to itself (e.g., USDT → USDT), this `continue` statement causes the function to skip the swap operation entirely. However, the function still completes successfully and returns `amountOut = 0`, while the tokens remain in the contract's balance.

The attack begins when a victim (`Alice`) transfers tokens to the `FundsManager` contract. Alice then calls the external `swapManyToOne` function with the same token as both input and output (e.g., USDT → USDT). Due to the `continue` statement in `_swapManyToOne`, the

function the external function transfers 0 tokens to Alice. Later, an attacker (Bob) can call `swapManyToOne` with a different output token (e.g., USDT → WBTC). Since the contract has Alice's USDT in its balance, Bob's swap operation will:

- Use Alice's stuck USDT as the input token
- Execute the swap through the router (USDT → WBTC)
- Receive WBTC tokens
- Drain Alice's USDT from the contract

The `multiWithdraw` tracks the balance before and after calling `_swapManyToOne`, then adds the difference to account for tokens that were skipped by the `continue` statement. The external `swapManyToOne` function lacks this balance tracking mechanism.

```
function multiWithdraw(MultiSwap calldata multiSwap, address leader, address to) external nonReentrant returns (uint256 amountOut) {
    uint256 balanceBefore = IERC20(multiSwap.token).balanceOf(address(this));
    (address[] memory tokens, uint256[] memory amounts) = vault.withdrawForSwap(leader, msg.sender);
    uint256 len = multiSwap.swaps.length;
    require(tokens.length == len, "Funds Manager: invalid token count");
    for (uint256 i = 0; i < len; i++) {
        require(tokens[i] == multiSwap.swaps[i].token, "Funds Manager: invalid token");
        require(amounts[i] == multiSwap.swaps[i].amount, "Funds Manager: invalid output amount");
    }
    uint256 balanceAfter = IERC20(multiSwap.token).balanceOf(address(this));
    amountOut = _swapManyToOne(multiSwap) + balanceAfter - balanceBefore;
    IERC20(multiSwap.token).safeTransfer(to, amountOut);
}
```

The same vulnerability also affects the `copyTrade()` function, which has even more severe consequences, as besides the mentioned issue it is making the vault's

accounting incorrect, i.e. leaders getting less tokens back from the fund manager:

```
function copyTrade(MultiSwap calldata multiSwap, address leader) external nonReentrant {
    require(msg.sender == tradeSigner, "Funds Manager: unauthorized");
    uint256 len = multiSwap.swaps.length;
    address[] memory tokens = new address[](len);
    uint256[] memory amounts = new uint256[](len);
    for (uint256 i = 0; i < len; i++) {
        tokens[i] = multiSwap.swaps[i].token;
        amounts[i] = multiSwap.swaps[i].amount;
    }
    vault.copyTradeWithdraw(leader, tokens, amounts);
    uint256 amountOut = _swapManyToOne(multiSwap);
    IERC20(multiSwap.token).safeTransfer(address(vault), amountOut);
    vault.copyTradeDeposit(leader, multiSwap.token, amountOut);
}
```

When a leader executes a copy trade that includes swapping a token to itself (e.g., USDT → USDT), the following sequence occurs:

1. Tokens are withdrawn from the vault via `vault.copyTradeWithdraw()`
2. The same-token swap is skipped in `_swapManyToOne()` due to the `continue` statement
3. Tokens get stuck in the funds manager instead of being returned to the vault
4. Vault accounting becomes incorrect `balances[leader][token]` is permanently reduced

PoC (Proof of Concept):

The vulnerability has been demonstrated with a test case:

```
it("Should demonstrate theft: Alice's USDT gets stolen by Bob", async () => {
    // Step 1: Alice transfers 100 USDT to FundsManager
```

```

const alice = follower;
const bob = follower2;
await usdt.connect(alice).transfer(fundsManager.getAddress(), ethers.parseUnits("100", 6));

// Step 2: Alice calls swapManyToOne with USDT → USDT (same token)
const aliceSwap = {
  router: await swapContract.getAddress(),
  token: await usdt.getAddress(), // Output token
  swaps: [
    {
      amount: ethers.parseUnits("100", 6), // 100 USDT
      token: await usdt.getAddress(), // Input token (same as output!)
      minAmountOut: 0n,
      swapData: "0x", // Empty swap data
    },
    deadline: ethers.MaxUint256,
  ];
  await fundsManager.connect(alice).swapManyToOne(aliceSwap, alice);

// Step 3: Bob steals Alice's USDT by swapping them to WBTC
const bobSwap = {
  router: await swapContract.getAddress(),
  token: await wbtc.getAddress(), // Output token (WBTC)
  swaps: [
    {
      amount: ethers.parseUnits("100", 6), // 100 USDT (Alice's stuck funds)
      token: await usdt.getAddress(), // Input token (USDT)
      minAmountOut: 0n,
      swapData: await wowmax.getSwap(await usdt.getAddress(), "100", await wbtc.getAddress()).then(r => r.data),
    },
    deadline: ethers.MaxUint256,
  ];
  await fundsManager.connect(bob).swapManyToOne(bobSwap, bob);
};

```

The test execution shows the complete attack flow:

Step 1 - Alice transfers 100 USDT to FundsManager

FundsManager USDT: 100000000

Alice USDT: 999999999000000000

Bob USDT: 1000000000000000000

Step 2 - Alice calls swapManyToOne(USDT→USDT)

After Alice's swap - FundsManager USDT: 100000000

After Alice's swap - Alice USDT: 999999999000000000

Alice received: 0

Step 3 - Bob steals Alice's USDT by swapping USDT→WBTC

Bob WBTC before: 1000000000000000000

FundsManager USDT before Bob's theft: 100000000

Bob WBTC after: 10000000000100000

FundsManager USDT after Bob's theft: 0

Bob stole WBTC worth: 100000

==== THEFT DEMONSTRATED ===

1. Alice transferred 100 USDT to FundsManager
2. Alice called swapManyToOne(USDT→USDT) and received 0 USDT
3. 100 USDT got stuck in the contract
4. Bob called swapManyToOne(USDT→WBTC) and stole Alice's USDT!
5. Alice lost 100 USDT, Bob gained WBTC worth 100 USDT

Recommendation:

Fix both the `swapManyToOne` and `copyTrade` functions to include proper balance tracking, like it is done in the `multiWithdraw`.

H-03. Zero minimum output swaps cause permanent loss and drainage of stuck

tokens

Status: Fixed

Customer's response:

Zero-output swaps remain intentionally allowed for the multiWithdraw method, as followers sometimes hold negligible token amounts. Tracking and transferring such values would be more costly in gas than the tokens' worth. During copy trades, such swaps cannot occur, since this is ensured at the backend. Nevertheless, additional checks were added to handle these cases defensively.

Severity: High

Path: [contracts/WowmaxCopyTradingFundsManager.sol#L207-L237](#)

Description:

The `WowmaxCopyTradingFundsManager.sol` contract contains a vulnerability in the `_swapManyToOne()` function. There is a `continue` statement that skips swaps when `minAmountOut == 0`, in addition to the existing same-token swap vulnerability. This creates another attack vector where funds become permanently stuck in the contract.

```
function _swapManyToOne(MultiSwap calldata multiSwap) private returns (uint256 amountOut) {
    require(allowedRouters.contains(multiSwap.router), "Funds Manager: router not allowed");
    require(multiSwap.deadline >= block.timestamp, "Funds Manager: expired");
    uint256 amountIn;
    uint256 swapAmountOut;
    address tokenIn;
    uint256 balanceBefore = IERC20(multiSwap.token).balanceOf(address(this));
    uint256 balanceAfter;
    uint256 len = multiSwap.swaps.length;
```

```

        for (uint256 i = 0; i < len; i++) {
            amountIn = multiSwap.swaps[i].amount;
            tokenIn = multiSwap.swaps[i].token;
            if (tokenIn == multiSwap.token) {
                continue;
            }
            IERC20(tokenIn).safeIncreaseAllowance(multiSwap.router, amountIn);
            (bool success,) = multiSwap.router.call(multiSwap.swaps[i].swapData);
            require(success, "Funds Manager: swap failed");
            balanceAfter = IERC20(multiSwap.token).balanceOf(address(this));
            swapAmountOut = balanceAfter - balanceBefore;
            require(swapAmountOut >= multiSwap.swaps[i].minAmountOut, "Funds
Manager: insufficient output amount");
            balanceBefore = balanceAfter;
            amountOut += swapAmountOut;
        }
        return amountOut;
    }

```

If a user transfers tokens to the contract and attempts to swap with `minAmountOut = 0`, the funds become stuck due to the new `continue` statement. Any other user can then come and swap those stuck tokens to another token, effectively stealing the original user's funds.

Impact 1: Public Swap Theft

The external `swapManyToOne()` function directly calls `_swapManyToOne()`. If a user transfers tokens to the `FundsManager` contract and then calls `swapManyToOne()` with `minAmountOut = 0`, the `_swapManyToOne()` function will return 0 (because the swap was skipped). Consequently, the `swapManyToOne()` function will transfer 0 tokens back to the user, leaving the original tokens permanently stuck in the `FundsManager` contract. Any other user can then call `swapManyToOne()` with a different output token, effectively stealing the original user's stuck funds.

```

function swapManyToOne(MultiSwap calldata multiSwap, address to) external
nonReentrant override returns (uint256 amountOut) {

```

```

require(to != address(0), "Funds Manager: invalid recipient");
amountOut = _swapManyToOne(multiSwap);
IERC20(multiSwap.token).safeTransfer(to, amountOut);
}

```

Impact 2: Copy Trading Permanent Loss and Vault Accounting Inconsistency

The `copyTrade()` function also utilizes `_swapManyToOne()`, leading to a more severe impact on the protocol's core functionality and accounting:

```

function copyTrade(MultiSwap calldata multiSwap, address leader) external nonReentrant {
    require(msg.sender == tradeSigner, "Funds Manager: unauthorized");
    uint256 len = multiSwap.swaps.length;
    address[] memory tokens = new address[](len);
    uint256[] memory amounts = new uint256[](len);
    for (uint256 i = 0; i < len; i++) {
        tokens[i] = multiSwap.swaps[i].token;
        amounts[i] = multiSwap.swaps[i].amount;
    }
    vault.copyTradeWithdraw(leader, tokens, amounts);
    uint256 amountOut = _swapManyToOne(multiSwap);
    IERC20(multiSwap.token).safeTransfer(address(vault), amountOut);
    vault.copyTradeDeposit(leader, multiSwap.token, amountOut);
}

```

When a leader executes a copy trade that includes a swap with `minAmountOut = 0`:

1. Tokens are withdrawn from the `WowmaxCopyTradingVault` via `vault.copyTradeWithdraw()`. At this point, the vault's internal `balances[leader][token]` is correctly reduced.
2. The zero `minAmountOut` swap is skipped within `_swapManyToOne()` due to the continue statement. `amountOut` will be 0 for this specific swap.

3. The `FundsManager` attempts to transfer `amountOut` (which is 0) back to the vault via `IERC20(multiSwap.token).safeTransfer(address(vault), amountOut);`. No tokens are actually transferred.
4. The `FundsManager` then calls `vault.copyTradeDeposit()` with `amountOut` (which is 0). This means the vault's `balances[leader][token]` is never re-credited for the tokens that were supposed to be swapped and redeposited.
5. This creates a permanent loss scenario where the leaders lose tokens from their portfolio, and the followers lose value from their investments.
6. The vault's internal accounting becomes inconsistent: The balances mapping in the vault no longer accurately reflects the tokens that should be managed for the leader.

The stuck funds can later be drained by anyone using the public `swapManyToOne` function, as described in Impact 1.

Recommendation:

Revert the transaction when `minAmountOut` is zero instead of skipping the swap with `continue`. If there is a legitimate need for zero minimum output, apply the same balance tracking fix used in `multiWithdraw` to both `swapManyToOne` and `copyTrade` functions, as suggested in the **H-02**.

L-01. Missing `_disableInitializers()` in upgradeable vault contract

Status: Fixed

Severity: Low

Path: [contracts/WowmaxCopyTradingVault.sol](#)

Description:

The `WowmaxCopyTradingVault.sol` contract is upgradeable but does not call `_disableInitializers()` in its constructor. In upgradeable contract patterns, this call is a best practice to prevent the implementation (logic) contract from being initialized directly. While this doesn't affect the proxy's behavior, it helps protect against accidental or malicious use of the implementation contract in isolation, especially in environments where both proxy and implementation contracts are visible, like block explorers.

Recommendation:

Consider adding the following line to the constructor:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

L-02. Use ERC7201 namespaced storage layouts or storage gaps to prevent storage collision

Status: Acknowledged

Customer's response:

This currently only affects the inability to add new fields to the PricesVerifier contract. The VaultState contract can be extended safely with new fields; existing ones will remain unchanged. The Hardhat upgradeability plugin provides additional safety checks. For these reasons, we consider the notice low risk and acceptable to defer.

Severity: Low

Path: [contracts/WowmaxCopyTradingVault.sol](#)

Description:

The `WowmaxCopyTradingVault.sol` contract is upgradeable but does not use ERC-7201 namespaced storage pattern. OpenZeppelin v5 upgradeable contracts use ERC-7201 to prevent storage collision between different versions of the contract.

This contracts should either use:

- ERC-7201 namespaced storage layouts:
<https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/AccessControlUpgradeable.sol#L60-L72>
- Storage gaps, though this is an older method:
<https://blog.openzeppelin.com/introducing-openzeppelin-contracts-5.0#Namespaced>

This creates a collision risks, i.e. future upgrades may overwrite existing storage slots.

Recommendation:

Use one of the above two techniques.

I-01. Use custom errors instead of string-based require statements

Status: Fixed

Severity: Informational

Description:

The entire project extensively uses require statements with string error messages instead of custom errors. This pattern is found across all major contracts including

`WowmaxCopyTradingVault.sol` , `WowmaxCopyTradingFundsManager.sol` , `PricesVerifier.sol` , and `Rescuer.sol` .

The usage of custom errors will save a lot of gas during deployment as well as save on code bytesize of the contract. Furthermore, custom errors are much clearer as they allow for parameter values, making debugging much easier.

Recommendation:

Replace all `require` statements with string errors with custom errors.

I-02. Redundant imports

Status: Fixed

Severity: Informational

Path: [contracts/WowmaxCopyTradingFundsManager.sol#L12](#),
[contracts/WowmaxCopyTradingVault.sol#L11](#)

Description:

Both `WowmaxCopyTradingVault.sol` and `WowmaxCopyTradingFundsManager.sol` contain redundant imports of `Ownable` contracts. The contracts are importing both the base `Ownable` contract and the `2Step` version, which is unnecessary since `Ownable2Step` already inherits from `Ownable`.

`WowmaxCopyTradingVault.sol` :

```
import {Ownable2StepUpgradeable} from "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol";
import {OwnableUpgradeable} from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol"; //@audit-info redundant import
```

`WowmaxCopyTradingFundsManager.sol` :

```
import "@openzeppelin/contracts/access/Ownable2Step.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol"; //@aud
```

it-info redundant import

Recommendation:

Remove the redundant imports.

I-03. Missing events in FundsManager contract

Status: Fixed

Customer's response:

New events were added to both FundsManager and Vault contracts. Events for methods that interact with the Vault were not duplicated in FundsManager, since Vault already emits all necessary events.

Severity: Informational

Path: contracts/WowmaxCopyTradingFundsManager.sol

Description:

The `WowmaxCopyTradingFundsManager.sol` contract performs financial operations but does not emit any events. This makes it impossible to track important activities like swaps, and configuration changes off-chain.

Recommendation:

Add events for all critical operations.

I-04. Missing stableCoin setter and immutability concerns

Status: Acknowledged

Customer's response:

The stable coin is effectively immutable. Given the separation of contract logic and state, we propose to leave the current implementation unchanged.

Severity: Informational

Path: [contracts/WowmaxCopyTradingFundsManager.sol](#),
[contracts/WowmaxCopyTradingVault.sol](#)

Description:

The `WowmaxCopyTradingFundsManager.sol` and `WowmaxCopyTradingVault.sol` contracts lacks a setter function for the `stableCoin` address, making it immutable after deployment. This design choice reduces operational flexibility and could lead to issues if the stablecoin needs to be upgraded or changed. Additionally, the current implementation doesn't use the immutable keyword, which would provide gas optimization and clearer intent.

```
constructor(address _vault, address _tradeSigner, address _stableCoin) Ownable(msg.sender) {
    vault = IWowmaxCopyTradingVault(_vault);
    tradeSigner = _tradeSigner;
    stableCoin = IERC20(_stableCoin); // Set only once here
}
```

Recommendation:

Add a setter function with proper access control and validation, or if the stablecoin should never change, use the immutable keyword in the

`WowmaxCopyTradingFundsManager.sol`.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Hakira or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.



About Hakira

Hakira is a cutting-edge security company operating for both Web3 and Web2 assets. We specialize in comprehensive security assessments, smart contract audits, and vulnerability detection across digital ecosystems.

Our team comprises seasoned security professionals, blockchain experts, and penetration testers who bring diverse expertise to every project we undertake. We employ a combination of manual code review, automated scanning, and innovative security methodologies to deliver thorough and actionable security insights.

Our mission: To illuminate unseen risks and simplify complex threat environments with precision, clarity, and purpose.

At Hakira, we believe that security is not just about finding vulnerabilities - it's about understanding the broader implications of these findings and providing practical solutions that enhance the overall security posture of our clients' projects.

CONTACT US

hakira.io/contact-us