# Security Audit Report for wmx.fi

BitBarrier Team

November 2023

## 1 About BitBarrier

BitBarrier is a small group of expert security engineers offering reviews and other security-related services to Web3 projects. Our team is experienced in all aspects of blockchain technology, including protocol design, smart contracts, and the Solidity compiler. BitBarrier has been offered bug bounty awards from major decentralized trading protocols like DYDX and GMX.
Reach out to us at `https://twitter.com/bit_barrier`.

## 2 Introduction

wmx.fi is an on-chain derivatives exchange protocol with leverage. It utilizes a pool similar to GMX where user's can deposit collateral that pays out trader's profits and earns trader's losses. The protocol allows users to place/cancel orders and uses an oracle for settling orders in batches.
*Disclaimer:* This security review is a time-specific snapshot and does not guarantee against hacks.

## 3 Risk Classification

| **Impact** | **Description** |
|---|---|
| High | Significant loss (10%) of assets or harm to most users. |
| Medium | Losses under 10% or affecting a subset of users. |
| Low | Minor losses or easily repairable issues. |

Table 1: Impact Classification

| **Likelihood** | **Description** |
|---|---|
| High | Almost certain or easy to perform. |
| Medium | Conditionally possible or incentivized. |
| Low | Unlikely or minimal incentive. |

Table 2: Likelihood Classification

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

Table 3: Risk Classification

| Severity | Action Required |
|---|---|
| Critical | Immediate fix required. |
| High | Must fix before deployment. |
| Medium | Should fix. |
| Low | Could fix. |

Table 4: Action Required for Severity Levels

# 4 Executive Summary

BitBarrier reviewed the wmx.fi contracts over a period of 14 days. A total of 11 issues were identified, categorized by risk levels and including gas optimizations and informational findings.

| Detail | Information |
|---|---|
| Project Name | wmx.fi |
| Repositories | `https://github.com/wmxfi/contracts` |
| Type of Project | DeFi, Perpetuals Exchange |
| Audit Timeline | Nov 15 2023 - Nov 30 2023 |

Table 5: wmx.fi Audit Summary

| Severity | Count |
|---|---|
| Critical Risk | 0 |
| High Risk | 2 |
| Medium Risk | 3 |
| Informational | 6 |
| **Total** | 11 |

Table 6: Issues Found in Audit

# 5 Findings

## 5.1 High Risk

### 5.1.1 Possible over-utilization in the pool

**Severity:** High Risk
**Context:** Pool.sol#L105
**Description:** The current implementation of the `withdraw` function does not check for pool utilization overflow. This oversight allows users to withdraw funds under unfavorable trading profit conditions, where the pool would be required to payout the trader's profits.
**Recommendation:** Amend the function to include a check for pool utilization.

```
function withdraw(uint256 currencyAmount) external {
    ...
+   uint256 utilization = getUtilization();
+   require(utilization < 10 ** 4, "!utilization");
    ....
}
```

**WmxFi:** Resolved with the above check added into the `withdraw` function
**BitBarrier:** Verified

#### 5.1.2 Incorrect refunds in cancelOrder

**Severity:** High Risk
**Context:** Trading.sol
**Description:** The `cancelOrder` function in wmx.fi's 'Trading.sol' contract does not differentiate between open and close orders. Since close orders do not require margin, this allows an attacker to exploit the system by submitting and immediately canceling a close order, leading to the improper refund of margin and fees. Repeated execution of this exploit can drain funds from the Trading Contract and enable the holding of large, risk-free positions.
**Recommendation:** Implement a check within the `cancelOrder` function to differentiate between open and close orders. This will prevent incorrect refunds and mitigate the risk of fund drainage and exploitation by attackers.

```
function cancelOrder(
    bytes32 productId,
    address currency,
    bool isLong
) external {
    bytes32 key = _getPositionKey(msg.sender, productId, currency, isLong);
    Order memory order = orders[key];
    require(order.size > 0, "!exists");
    Product memory product = products[productId];
    uint256 fee = order.size * product.fee / 10**6;
    _updateOpenInterest(currency, order.size, true);
    delete orders[key];
    // Refund margin + fee
    uint256 marginPlusFee = order.margin + fee;
    _transferOut(currency, msg.sender, marginPlusFee);
}
```

**WmxFi:** Resolved with different code paths for open/close orders
**BitBarrier:** Verified

### 5.2 Medium Risk

#### 5.2.1 Order Overwrite in submitOrder Function

**Severity:** Medium Risk
**Context:** Trading.sol
**Description:** The `submitOrder` function in WMX's smart contract allows pending orders to be overwritten. This is not an intended usecase as end-users might assume that both orders are active or executed.
**Recommendation:** Enhance the order validation logic in the `submitOrder` function to check for existing pending orders.

```
function submitOrder(
        bytes32 productId,
        address currency,
        bool isLong,
        uint256 margin,
        uint256 size
    ) external payable {
    ...
    bytes32 key = _getPositionKey(msg.sender, productId, currency, isLong);
    Order memory order = orders[key];
+   require(order.size == 0, "!order");
    Product memory product = products[productId];
```

```
    uint256 fee = size * product.fee / 10**6;
    ...
```

**WmxFi:** Resolved with the above check added into the function
**BitBarrier:** Verified

#### 5.2.2 Centralization Risk and Oracle Manipulation

**Severity:** Medium Risk
**Context:** Oracle.sol
**Description:** Functions `settleOrder` and `liquidatePosition` can only be executed by the dark oracle address. These functions depend on an external off-chain oracle for price feeds. Currently, there are no safeguards to ensure the prices are within reasonable bounds or delay intervals. This could lead to inaccurate settlements and liquidations due to potential oracle manipulation or delays.

```
function settleOrders(
    address[] calldata users,
    bytes32[] calldata productIds,
    address[] calldata currencies,
    bool[] calldata directions,
    uint256[] calldata prices
) external onlyDarkOracle
```

**Recommendation:** Implement checks to validate oracle data against predefined bounds or introduce delay intervals.
**WmxFi:** The team accepts this risk as it is the defining logic of the exchange. We will move to a Pyth oracle based pricing in our v2 iteration.

#### 5.2.3 Product Parameter Validation

**Severity:** Medium Risk
**Context:** Trading.sol#L280
**Description:** Current implementation of functions `addProduct` and `updateProduct` lacks comprehensive validation on parameters except for the liquidation threshold. This can lead to misconfigurations involving leverage, fees, and interests. There is a centralization risk, as the owner can update the product liquidation threshold, potentially leading to forced liquidation of users or unjustified fee adjustments.
**Recommendation:** Implement additional checks on all product parameters during setting and updating processes.
**WmxFi:** The team accepts the centralization risk in this iteration of the product. More checks will be added in v2.

### 5.3 Informational

#### 5.3.1 Pool Utilization Limits are hard-coded

**Severity:** Informational
**Context:** Trading.sol#L201
**Description:** The check for pool utilization is hard-coded here - `require(utilization < 10**4, "!utilization");` In the future, it would be necessary to limit utilization for different collateral assets depending on volatility.
**Recommendation:** Add a configurable parameter in the pool to check against.

#### 5.3.2 Unnecessary Underflow Checks

**Severity:** Informational
**Context:** Trading.sol#L185

**Description:** Solidity 0.8.0, by default, reverts when there is an overflow or underflow in an operation on integers. There is no need to add an extra check here
**Recommendation:** Remove the underflow check

```
function submitOrder(
        bytes32 productId,
        address currency,
        bool isLong,
        uint256 margin,
        uint256 size
    ) external payable {
    ...
-   require(margin > fee, "!margin<fee");
    margin -= fee;
    ...
```

#### 5.3.3 Unchecked External Calls

**Severity:** Informational
**Context:** Trading.sol#426, Rewards.sol#101 and Pool.sol#143
**Description:** The presence of `fallback()` and `receive()` functions in the smart contract. These functions currently lack limitations on who can send funds or under what conditions funds can be received. This oversight could result in an unintentional WEMIX balance within the contract.
**Recommendation:** It's advised to either restrict or remove these functions, or implement appropriate handling mechanisms to mitigate the risk.

#### 5.3.4 Stricter Input Validation

**Severity:** Informational
**Description:** The contract currently performs minimal checks such as `!size` and `!margin`. However, these checks are not comprehensive enough to fully cover the business logic requirements.
**Recommendation:** Implement more fine-grained input validation to ensure all aspects of the business logic are correctly handled and secure.

#### 5.3.5 Make Pool.currency immutable

**Severity:** Informational
**Context:** Pool.currency variable in src/Pool.sol at line 18.
**Description:** The Pool.currency variable is currently not set as immutable, which could lead to potential security risks or unintended behavior in the smart contract. We do not see any reason to be switching pool currencies once initialised as they would be needed for deposits and withdrawals.
**Recommendation:** Modify the Pool.currency variable to be immutable, enhancing the security and reliability of the contract.

#### 5.3.6 Usage of Floating Pragma

**Severity:** Informational
**Description:** The contract utilizes a floating pragma `^0.8.0`, which poses a potential safety risk. Floating pragma can lead to unpredictability in behavior when compiling with future Solidity versions that may be incompatible.
**Recommendation:** It is advised to pin the Solidity version to a fixed version or to a more constrained range. This change will enhance stability and predictability in the contract's behavior.

# 6 Slither Analysis

### 6.0.1 Command

```
slither .
    - 'forge clean' running (wd: ~/wmxfi-contracts)
    - 'forge build --build-info --skip */test/** */script/** --force' running (wd: ~/wmxfi-contracts)
```

### 6.0.2 Reentrancy reports

```
Reentrancy in Pool.deposit(uint256) (src/Pool.sol#81-103)
Reentrancy in Trading.liquidatePosition(address,bytes32,address,bool,uint256) (src/Trading.sol#352-389)
Reentrancy in Trading.submitCloseOrder(bytes32,address,bool,uint256) (src/Trading.sol#190-229)
Reentrancy in Trading.submitOrder(bytes32,address,bool,uint256,uint256) (src/Trading.sol#140-188)
Reentrancy in Trading.releaseMargin(address,bytes32,address,bool,bool)
```

All the reports above are either privileged functions or restricted calls on authorized token contracts.

### 6.0.3 Loss of precision

```
Pool.withdraw(uint256) (src/Pool.sol#105-134) performs a multiplication on the result of
a division:
        - amount = currencyAmount * totalSupply / currentBalance (src/Pool.sol#113)
        - currencyAmount = amount * currentBalance / totalSupply (src/Pool.sol#116)
Trading.submitCloseOrder(bytes32,address,bool,uint256) (src/Trading.sol#190-229) performs a
multiplication on the result of a division:
        - fee = size * product.fee / 10 ** 6 (src/Trading.sol#206)
        - fee_units = fee * 10 ** (18 - UNIT_DECIMALS) (src/Trading.sol#208)
```

`currencyAmount` and `size` are always limited by the amount users are able to provide as margin multiplied by the leverage and therefore, is well within the precision limits.

### 6.0.4 Strict-Equality Checks

```
Pool._transferOut(address,uint256) (src/Pool.sol#153-162) uses a dangerous strict equality:
        - amount == 0 || to == address(0) (src/Pool.sol#154)
Pool.deposit(uint256) (src/Pool.sol#81-103) uses a dangerous strict equality:
        - lastBalance == 0 || totalSupply == 0 (src/Pool.sol#91)
Pool.getCurrencyBalance(address) (src/Pool.sol#181-185) uses a dangerous strict equality:
        - totalSupply == 0 (src/Pool.sol#182)
Pool.getUtilization() (src/Pool.sol#175-179) uses a dangerous strict equality:
        - currentBalance == 0 (src/Pool.sol#177)
Trading._transferOut(address,address,uint256) (src/Trading.sol#446-455) uses a dangerous
strict equality:
        - amount == 0 || to == address(0) (src/Trading.sol#447)
```

The above checks are expected initial-state validations.

# 7 Test Coverage

## 7.1 Unit Tests

### 7.1.1 Trading.sol

```
[PASS] testSetAndChangeOwner()
[PASS] testUnauthorizedUserCannotSetOwner()
[PASS] testAddNewProduct()
[PASS] testAddExistingProduct()
[PASS] testUpdateProduct()
[PASS] testUpdateNonExistingProduct()
[PASS] testSuccessfulOrderSubmission()
[PASS] testSuccessfulCloseOrderSubmission()
[PASS] testFailToCloseNonExistingPosition()
[PASS] testCancelExistingOrder()
```

```
[PASS] testFailToCancelNonExistingOrder()
[PASS] testSettleOrderByOracle()
[PASS] testLiquidatePositionByOracle()
[PASS] testDistributeFeesToTreasury()
[PASS] testFailDistributeFeesByNonOwner()
[PASS] testReleaseMarginForPosition()
[PASS] testFailReleaseMarginByNonOwner()
[PASS] testValidatePriceInOrderSettlement()
[PASS] testSubmitOrderInsufficientMargin()
[PASS] testSubmitOrderZeroSize()
[PASS] testCloseOrderSizeGreaterThanPosition()
[PASS] testCloseNonExistentPosition()
[PASS] testSubmitOrderWithLowMargin()
[PASS] testSubmitOrderWithInvalidLeverage()
[PASS] testSubmitOrderWithExcessiveFee()
[PASS] testLiquidateNonExistentPosition()
```

### 7.1.2 Oracle.sol

```
[PASS] testChangeOwner()
[PASS] testUnauthorizedChangeOfOwner()
[PASS] testSetRouter()
[PASS] testSetOracleParameters()
[PASS] testSettleOrders()
[PASS] testLiquidatePositions()
```

### 7.1.3 Pool.sol

```
[PASS] testDeposit()
[PASS] testDepositExceedingMaxCap()
[PASS] testWithdraw()
[PASS] testRevertInsufficientDeposit()
[PASS] testRevertInsufficientWithdrawal()
[PASS] testWithdrawBeforeCooldown()
[PASS] testWithdrawMoreThanBalance()
[PASS] testEmitDepositEvent()
[PASS] testEmitWithdrawEvent()
[PASS] testDepositZeroAmount()
[PASS] testWithdrawZeroAmount()
[PASS] testSetOwner()
[PASS] testUnauthorizedUserCannotSetOwner()
[PASS] testUpdateOpenInterestIncrease()
[PASS] testUpdateOpenInterestDecrease()
[PASS] testWithdrawWithFeeDeduction()
[PASS] testSetRouter()
[PASS] testUnauthorizedUserCannotSetRouter()
[PASS] testSetParameters()
[PASS] testUnauthorizedUserCannotSetParameters()
[PASS] testDepositExceedsMaxCapAfterMultipleDeposits()
[PASS] testWithdrawFeeCalculation()
[PASS] testUtilizationEffectOnWithdrawal()
```