Frontrun Market Security Review

Pashov Audit Group

Conducted by: T1MOH, Dan Ogurtsov

February 29th 2024 - March 4th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Introduction

A time-boxed security review of the **frontrun-market/contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Frontrun Market

A decentralised peer to peer OTC trading platform to trade Points, Airdrop Allocations, Pre-Market tokens and NFT Whitelists. Users can sell or buy points or airdrop allocations by depositing a collateral and upon settlements of orders it is returned back to the sellers/buyers.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. 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.

5.3. Action required for severity levels

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

6. Security Assessment Summary

review commit hash - 4fe59a4a1c69a6accb99e32cbb044438077eeb2d

fixes review commit hash - 35c836d5ea1c5ffd2aa68762a856d2837d8b4853

Scope

The following smart contracts were in scope of the audit:

- `FrontrunMarket`
- `FrontrunMarketBlast`

7. Executive Summary

Over the course of the security review, T1MOH, Dan Ogurtsov engaged with Frontrun Market to review Frontrun Market. In this period of time a total of **9** issues were uncovered.

Protocol Summary

Protocol Name	Frontrun Market
Repository	https://github.com/frontrun-market/contracts
Date	February 29th 2024 - March 4th 2024
Protocol Type	OTC trading platform

Findings Count

Severity	Amount
Medium	2
Low	7
Total Findings	**9**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Excessive msg.value lost when creating an offer	Medium	Resolved
[M-02]	ETH transfer griefing	Medium	Acknowledged
[L-01]	Two order statuses share the same value	Low	Resolved
[L-02]	Operator role not used	Low	Resolved
[L-03]	Fee on transfer tokens not supported	Low	Acknowledged
[L-04]	batchFillOffer() checking the same lengths provided	Low	Resolved
[L-05]	pledgeRate setup not checked	Low	Acknowledged
[L-06]	Incorrect variable used to check order status	Low	Resolved
[L-07]	Offer with incorrect offerType is treated as sell offer	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Excessive `msg.value` lost when creating an offer

Severity

Impact: High, funds lost forever

Likelihood: Low, requires a mistake from a user

Description

`msg.value` is considered valid if it is greater or equal than needed:

```
function newOfferETH(
    uint8 offerType,
    bytes32 tokenId,
    uint256 amount,
    uint256 value,
    uint256 minAmount
  ) external payable nonReentrant {
    ...

    // collateral
    uint256 collateral = (value * $.config.pledgeRate) / WEI6;

    uint256 _ethAmount = offerType == OFFER_BUY ? value : collateral;
@>  require(_ethAmount <= msg.value, 'Insufficient Funds');
    ...
  }
```

But the amount to pay is static and only depends on submitted offer params, so excessive `msg.value` means a mistake on the user's side. For example, software performing trading contains an error.

Recommendations

Don't accept excessive `msg.value`:

```
-   require(_ethAmount <= msg.value, 'Insufficient Funds');
+   require(_ethAmount == msg.value, 'Insufficient Funds');
```

[M-02] ETH transfer griefing

Severity

Impact: High

Likelihood: Low

Description

`forceCancelOrder()` uses a direct address call to send ETH to buyer and seller on refund in ETH.

```
if (offer.exToken == address(0)) {
      // refund ETH
      if (buyerRefundValue > 0 && buyer != address(0)) {
        (bool success,) = buyer.call{value: buyerRefundValue}('');
        require(success, 'Transfer Funds to Seller Fail');
      }
      if (sellerRefundValue > 0 && seller != address(0)) {
        (bool success,) = seller.call{value: sellerRefundValue}('');
        require(success, 'Transfer Funds to Seller Fail');
      }
```

Both buyer and seller can grief each other reverting on these calls, blocking refund execution for both parties and managing the suitable time/conditions to finalize the refund. In the worst cases, it can be used as a means of blackmailing. In addition, it can be not intentional reverts, e.g. when the receiver is a smart-contract that hasn't designed refund logic (and cannot receive ETH).

Recommendations

Consider implementing a claiming logic when users do not receive calls and transfers, and the contract stores their pending withdrawals. Users have to use a separate function to claim these funds. This logic is worth implementing for all ETH transfers - including `settleFilled()`, `settle2Steps()` and `cancelOffer()`.

8.2. Low Findings

[L-01] Two order statuses share the same value

`STATUS_ORDER_CANCELLED` and `STATUS_ORDER_SETTLE_CANCELLED` both have the same value = 3.

```
uint8 private STATUS_ORDER_SETTLE_CANCELLED = 3;
  uint8 private STATUS_ORDER_CANCELLED = 3;
```

It means that the getters for cancel status will not be able to differentiate between these two statuses.

Consider either leaving only one cancel status or setting STATUS_ORDER_CANCELLED = 4.

[L-02] Operator role not used

According to comments these functions were expected to check the `OPERATOR_ROLE`: `forceCancelOrder()` - now checks onlyOwner `settle2Step()` - now checks onlyOwner `settleCalcelled()` - 'Buyer or Operator Only', but now checks buyer and onlyOwner

As a result, the operator role is not used but exists as a variable.

Consider checking the operator in the above functions or remove the role.

[L-03] Fee on transfer tokens not supported

`acceptedTokens` can receive any token as collateral. In fact, not all of them can be used. E.g. fee-on-transfer and rebasable tokens can stuck, as the contract will always receive less balance than provided for transferFrom() with the following problem to distribute them.

Ensure that no such tokens are accepted.

[L-04] batchFillOffer() checking the same lengths provided

Consider checking `offerId.length == amount.length` in `batchFillOffer()`, same way as it is done in `settle2StepsBatch()`.

[L-05] pledgeRate setup not checked

`pledgeRate` setup accepts any number without checks in `updateConfig()`. Consider adding sanity checks for setup.

[L-06] Incorrect variable used to check order status

As you can see status of order is checked against offer's variable. Now they are the same, but mistakes can have an impact in the future

```
// Status
  // Offer status
  uint8 private STATUS_OFFER_OPEN = 1;
  uint8 private STATUS_OFFER_FILLED = 2;
  uint8 private STATUS_OFFER_CANCELLED = 3;

  // Order Status
  uint8 private STATUS_ORDER_OPEN = 1;
  uint8 private STATUS_ORDER_SETTLE_FILLED = 2;
  uint8 private STATUS_ORDER_SETTLE_CANCELLED = 3;
  uint8 private STATUS_ORDER_CANCELLED = 3;

  function forceCancelOrder(uint256 orderId) public nonReentrant onlyOwner {
    MarketStorage storage $ = _getOwnStorage();
    Order storage order = $.orders[orderId];
    Offer storage offer = $.offers[order.offerId];

@>  require(order.status == STATUS_OFFER_OPEN, 'Invalid Order Status');
    ...
  }
```

[L-07] Offer with incorrect `offerType` is treated as sell offer

There are no checks on `offerType` the during offer creation. So if user mistakenly sets an incorrect `offerType` to an arbitrarily value, it's treated as a sell offer:

```
uint8 private OFFER_BUY = 1;
  uint8 private OFFER_SELL = 2;

  function newOffer(
    uint8 offerType,
    bytes32 tokenId,
    uint256 amount, //amount of asset
    uint256 value, // amount of collateral
    address exToken,
    uint256 minAmount
  ) external nonReentrant {
    ...

    // transfer offer value (offer buy) or collateral (offer sell)
@>  uint256 _transferAmount = offerType == OFFER_BUY ? value : collateral;
    iexToken.safeTransferFrom(msg.sender, address(this), _transferAmount);

    ...
  }
```

It is recommended to add sanity check:

```
require(token.status == STATUS_TOKEN_ACTIVE, 'Invalid Token');
    require(exToken != address
      (0) && $.acceptedTokens[exToken], 'Invalid Offer Token');
    require(amount > 0 && value > 0, 'Invalid Amount or Value');
    require(
      amount>=minAmount,
      'minamounttobefilledcantbegreaterthenamount'
    );
    require(
      minAmount>=token.minAmount,
      'minamountshouldbegreatertheneualtomarketglobalminamount'
    );
+   require(offerType == OFFER_BUY || offerType == OFFER_SELL);
```