SHERLOCK

Security Review For Ethos Network

Public Audit Contest Prepared For:	**Ethos Network**
Lead Security Expert:	**bughuntoor**
Date Audited:	**December 27 - December 30, 2024**
Final Commit:	**e3f40c9**

Introduction

Ethos Network is an onchain social credibility platform. This contest focuses on Reputation Markets - a new way to evaluate and trade on credibility.

Scope

Repository: trust-ethos/ethos

Audited Commit: 9c41a570cb87fc19989a6880ae2d6a21b3d36c1b

Final Commit: e3f40c932e38e2e3a0e005a074e71ea953aa432d

Files:

- packages/contracts/contracts/ReputationMarket.sol
- packages/contracts/contracts/errors/ReputationMarketErrors.sol
- packages/contracts/contracts/utils/LMSR.sol

Final Commit Hash

e3f40c932e38e2e3a0e005a074e71ea953aa432d

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
0	2

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0xaxaxa	X12	theodore
0xnegan	bughuntoor	tnquanghuy0512
Al-Qa-qa	dobrevaleri	whitehair0330
Benterkii	future2_22	zxriptor
JohnTPark24	moray5554	

Issue M-1: Determining how many votes to buy may run OOG.

Source: https://github.com/sherlock-audit/2024-12-ethos-update-judging/issues/43

Found by

bughuntoor

Summary

The current way buying votes works is that a user sends a certain `msg.value` and a minimum and maximum amount they wish to buy, and the contract loops through the values to find the maximum amount the user can actually buy.

```
(, , , uint256 total) = _calculateBuy(markets[profileId], isPositive,
↪  minVotesToBuy);
if (total > msg.value) revert InsufficientFunds();

(
  uint256 purchaseCostBeforeFees,
  uint256 protocolFee,
  uint256 donation,
  uint256 totalCostIncludingFees
) = _calculateBuy(markets[profileId], isPositive, maxVotesToBuy);
uint256 currentVotesToBuy = maxVotesToBuy;
// if the cost is greater than the maximum votes to buy,
// decrement vote count and recalculate until we identify the max number of votes
↪  they can afford
while (totalCostIncludingFees > msg.value) {
  currentVotesToBuy--;
  (purchaseCostBeforeFees, protocolFee, donation, totalCostIncludingFees) =
↪  _calculateBuy(
    markets[profileId],
    isPositive,
    currentVotesToBuy
  );
}
```

The problem is that this way is highly gas inefficient. And even though protocol is to be deployed on Base where gas costs are low, it would still be possible to reach significant gas costs.

Looping to check a certain buy's gas costs, costs around ~33k gas (PoC attached below). Considering users can easily be buying tens of thousands of votes, these checks could possibly reach the Base gas limit, which would not only make the tx impossible to execute, but even if it was possible it would cost a substantial amount of funds. As the gas limit is 240,000,000, it would require ~7000 iterations to run OOG.

Root Cause

Gas inefficient system design.

Attack Path

1. There is a market with base price of 0.0001 eth.
2. The market is one-sided and a Trust vote costs 1/10th of that - 0.00001 eth.
3. A user wants to buy votes for 1 ETH. That would be 100,000 votes.
4. User applies 10% slippage, this makes the `minVotesToBuy` 90,000.
5. Due to the price movements, the user can only buy 91,000 votes. As this requires 9,000 iterations, the tx will run OOG, as it cannot fit in a single Base block.
6. In the scenario where the gas required is just enough to fit in a block, this would cost the user `240_000_000 * 0.12e9 = 0.0288e18 = 0.0288 ETH`. (assuming historical base gas costs of 0.12 gwei). At current prices this is ~$100.

Impact

Impossible to execute certain transactions due to OOG. High gas costs.

Affected Code

https://github.com/sherlock-audit/2024-12-ethos-update/blob/main/ethos/packages/contracts/contracts/ReputationMarket.sol#L460

PoC

As the codebase lacked a Foundry test suite, I integrated the necessary contract parts in my own test suite.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";
import "../src/LMSR.sol";

contract MyTest is Test {
    uint256 basePrice = 1e18;
    struct Market {
        uint256[2] votes;
    }

    function test_gasCalc() public {
        uint256 initYesVotes = 50;
        uint256 initNoVotes = 30;

        Market memory testMarket;
        testMarket.votes[0] = initYesVotes;
        testMarket.votes[1] = initNoVotes;

        uint256 gasLeft = gasleft();

        uint256[] memory voteDelta = new uint256[](2);

        voteDelta[0] = testMarket.votes[0];
        voteDelta[1] = testMarket.votes[1] + 1;

        int256 costRatio = LMSR.getCost(
            initYesVotes,
            initNoVotes,
            voteDelta[0],
            voteDelta[1],
            1000);


        uint256 positiveCostRatio = costRatio > 0 ? uint256(costRatio) :
 ↪  uint256(costRatio * -1 );

        uint256 cost = positiveCostRatio * basePrice / 1e18;

        uint256 gasLeft2 = gasleft();
        uint256 gasConsumed = gasLeft - gasLeft2;
        console.log(gasConsumed);
    }
}
```

Mitigation

Consider using a binary search.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2389

Issue M-2: Incorrect rounding in the `ReputationMarket._calcCost()` function.

Source: https://github.com/sherlock-audit/2024-12-ethos-update-judging/issues/66

Found by

0xaxaxa, 0xnegan, Al-Qa-qa, Benterkii, JohnTPark24, X12, bughuntoor, dobrevaleri, future2_22, moray5554, theodore, tnquanghuy0512, whitehair0330, zxriptor

Summary

When buying and selling, the cost is calculated using rounding based on `isPositive`. However, the rounding should be based on `isBuy`, not `isPositive`.

Root Cause

The _calcCost() function calculates `cost` by rounding based on `isPositive`.

If `isPositive` is `false` (indicating that `DISTRUST` votes are being traded), the calculation is rounded up.

Consider the following scenario:

1. A user buys 2 `DISTRUST` votes.
2. The user sells a `DISTRUST` vote.
3. The user sells another `DISTRUST` vote.

During the buying process, rounding up occurs once, but when selling, rounding up occurs twice—at steps 2 and 3. As a result, `marketFunds` will be decremented by a dust amount.

If `marketFunds` was originally 0 (with the market created by the admin at a 0 creation cost), then step 3 becomes impossible.

In fact, `isPositive` is never related to the rounding direction.

```
      function _calcCost(
        Market memory market,
        bool isPositive,
        bool isBuy,
        uint256 amount
      ) private pure returns (uint256 cost) {
        ...

        int256 costRatio = LMSR.getCost(
          market.votes[TRUST],
          market.votes[DISTRUST],
          voteDelta[0],
          voteDelta[1],
          market.liquidityParameter
        );

        uint256 positiveCostRatio = costRatio > 0 ? uint256(costRatio) :
↪   uint256(costRatio * -1);
        // multiply cost ratio by base price to get cost; divide by 1e18 to apply
↪   ratio
        cost = positiveCostRatio.mulDiv(
          market.basePrice,
          1e18,
1057      isPositive ? Math.Rounding.Floor : Math.Rounding.Ceil
        );
      }
```

Internal pre-conditions

External pre-conditions

Attack Path

Impact

The last vote might not be sold.

PoC

Mitigation

Use `!isBuy` instead of `isPositive`.

```
        cost = positiveCostRatio.mulDiv(
          market.basePrice,
          1e18,
-         isPositive ? Math.Rounding.Floor : Math.Rounding.Ceil
+         !isBuy ? Math.Rounding.Floor : Math.Rounding.Ceil
        );
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2387

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.