Security Review For Ethos Network

Public Audit Contest Prepared For:	**Ethos Network**
Lead Security Expert:	**bughuntoor**
Date Audited:	**November 29 - December 5, 2024**
Final Commit:	**577fb40**

Introduction

Ethos Network is an onchain social reputation platform. This contest focuses on the financial stakes: vouching for others, participating in reputation markets. This builds on existing Ethos Network social contracts.

Scope

Repository: trust-ethos/ethos

Audited Commit: 12e6a9d3b8130404632664837 33a84218a35847f

Final Commit: 577fb400ea0aa022e19bcc670d2ad2b7cb2af183

Files:

- packages/contracts/contracts/EthosVouch.sol
- packages/contracts/contracts/ReputationMarket.sol
- packages/contracts/contracts/errors/ReputationMarketErrors.sol
- packages/contracts/contracts/utils/Common.sol

Final Commit Hash

577fb400ea0aa022e19bcc670d2ad2b7cb2af183

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
4	3

Security experts who found valid issues

0rpse
0x486776
0xAnmol
0xDemon
0xEkko
0xKann
0xMosh
0xPhantom2
0xProf
0xaxaxa
0xbakeng
0xc0ffEE
0xgremlincat
0xlucky
0xmujahid002
0xpiken
1337web3
4th05
Abhan1041
AestheticBhai
Al-Qa-qa
Artur
Bbash
BengalCatBalu
Bozho
Ch_301
Contest-Squad
Cybrid
DenTonylifer
DharkArtz
DigiSafe
HOM1T
John44
JohnTPark24
KingNFT
KlosMitSoss
Kyosi
LeFy
Limbooo
MohammadX2049
Nave765
NickAuditor2
POB
Pablo
Pheonix
Ryonen
SovaSlava
Waydou
Weed0607
X0sauce
X12
ami
aycozyOx
blutorque
bube
bughuntoor
chaos304
copperscrewer
cryptic
debugging3
dobrevaleri
durov
eLSeR17
farismaulana
future2_22
hals
iamnmt
iamthesvn
immeas
irresponsible
justAWanderKid
ke1caM
kenzo123
mahdikarimi
moray5554
newspacexyz
nikhilx0111
novaman33
parzival
pashap9990
qandisa
redbeans
rmdanxyz
rscodes
rudhra1749
shui
smbv-1923
t.aksoy
t0x1c
tachida2k
tjonair
tmotfl
tobi0x18
udo
underdog
vatsal
volodya
wellbyt3
whitehair0330
y4y
ydlee
zeGarcao
zhenyazhd
zxriptor

Issue H-1: Bonding curve logic can be exploited to pay less for buying votes

Source: https://github.com/sherlock-audit/2024-11-ethos-network-ii-judging/issues/167

Found by

Al-Qa-qa, LeFy, MohammadX2049, X12, ami, bughuntoor, future2_22, newspacexyz, t0x1c, tobi0x18, zxriptor

Description & Impact

The _calcVotePrice() function uses the following bonding curve formula:

```
File: ethos/packages/contracts/contracts/ReputationMarket.sol

912:            /**
913:             * @notice Calculates the buy or sell price for votes based on market state
914:@--->        * @dev Uses bonding curve formula: price = (votes * basePrice) / totalVotes
915:             * Markets are double sided, so the price of trust and distrust votes always sum to the base price
916:             * @param market The market state to calculate price for
917:             * @param isPositive Whether to calculate trust (true) or distrust (false) vote price
918:             * @return The calculated vote price
919:             */
920:            function _calcVotePrice(Market memory market, bool isPositive) private pure returns (uint256) {
921:              uint256 totalVotes = market.votes[TRUST] + market.votes[DISTRUST];
922:              return (market.votes[isPositive ? TRUST : DISTRUST] * market.basePrice) / totalVotes;
923:            }
```

This can be manipulated in the following ways:

Example1: (*coded in PoC1*)

- Normal Scenario:
    - For a default market config, Alice decides to buy some trust votes.
    - She calls `buyVotes()` with `0.1 ETH` of funds.
    - This fetches her 12 trust votes and costs exactly `0.098439322450225045 ETH`.

- Attack Scenario:
  - For a default market config, Alice decides to buy 12 trust votes.
  - She decides to alternate her `buyVotes()` call between 1 trust vote and 1 distrust vote.
  - She repeats this 12 times.
  - She sells all her distrust votes at the end.
  - She ends up paying `0.079440616542537061 ETH`, which is `0.02 ETH` lesser than the normal scenario.

In fact in general, if Alice wishes to buy a higher-priced vote then it works in her favour to buy the lower-priced one first to make the vote ratio 1:1. At the end, this lower-priced purchase can be sold off for a net gain.

Example2: (*coded in PoC2*)

- Normal Scenario:
  - For a default market config, Bob is sitting with 99 trust votes he bought a few moments ago.
  - Alice wants to buy 100 trust votes. She calls `buyVotes()` with `1 ETH` of funds.
  - This fetches her 100 trust votes and costs exactly `0.993452851893538135 ETH`.
- Attack Scenario:
  - For a default market config, Bob is sitting with 99 trust votes he bought a few moments ago.
  - Alice wants to buy 100 trust votes.
  - She first buys 99 distrust votes.
  - She now buys the 100 trust votes.
  - She then sells all her distrust votes.
  - She ends up paying `0.711616420426520863 ETH`, which is `0.28 ETH` lesser than the normal scenario.

Proof of Concept

Add this file as `rep.bondingCurveManipulation.test.ts` inside the `ethos/packages/contracts/test/reputationMarket/` directory and run with `npm run hardhat -- test --grep "BuyVotes Cost Manipulation"` to see the output:

```
import { loadFixture } from '@nomicfoundation/hardhat-toolbox/network-helpers.js';
import { expect } from 'chai';
import hre from 'hardhat';
import { type ReputationMarket } from '../../typechain-types/index.js';
import { createDeployer, type EthosDeployer } from '../utils/deployEthos.js';
```

```
import { type EthosUser } from '../utils/ethosUser.js';
import { DEFAULT, MarketUser } from './utils.js';

const { ethers } = hre;

describe('BuyVotes Cost Manipulation', () => {
    let deployer: EthosDeployer;
    let ethosUserA: EthosUser;
    let alice: MarketUser;
    let reputationMarket: ReputationMarket;
    let market: ReputationMarket.MarketInfoStructOutput;

    beforeEach(async () => {
        deployer = await loadFixture(createDeployer);

        if (!deployer.reputationMarket.contract) {
        throw new Error('ReputationMarket contract not found');
        }
        ethosUserA = await deployer.createUser();
        await ethosUserA.setBalance('100');

        alice = new MarketUser(ethosUserA.signer);

        reputationMarket = deployer.reputationMarket.contract;
        DEFAULT.reputationMarket = reputationMarket;
        DEFAULT.profileId = ethosUserA.profileId;
        await reputationMarket
            .connect(deployer.ADMIN)
            .setUserAllowedToCreateMarket(DEFAULT.profileId, true);
        await reputationMarket.connect(alice.signer).createMarket({ value:
↪   ethers.parseEther('0.02') });
        market = await reputationMarket.getMarket(DEFAULT.profileId);
        expect(market.profileId).to.equal(DEFAULT.profileId);
        expect(market.trustVotes).to.equal(1);
        expect(market.distrustVotes).to.equal(1);
    });

    it('Normal Buy', async () => {
        // Record initial state
        const initialMarket = await
↪   reputationMarket.getMarket(ethosUserA.profileId);
        expect(initialMarket.trustVotes).to.equal(1n);
        expect(initialMarket.distrustVotes).to.equal(1n);

        // Record Alice's balance initially
        const aliceBalanceBefore = await ethosUserA.getBalance();

        // Alice buys trust votes
        const aliceInvestment = ethers.parseEther('0.1');
        await alice.buyVotes({
```

```
            buyAmount: aliceInvestment,
            expectedVotes: 12n,  // We know this from simulation
            slippageBasisPoints: 0  // 0% slippage allowed
        });

        // Verify Alice's position
        const alicePosition = await alice.getVotes();
        console.log("\n");
        console.log("votes position (TRUST)    =", alicePosition.trustVotes);
        expect(alicePosition.trustVotes).to.equal(12n);
        expect(alicePosition.distrustVotes).to.equal(0n);

        // Record Alice's balance now
        const aliceBalanceAfter = await ethosUserA.getBalance();

        // Log the total cost to Alice
        console.log('Cost1:', ethers.formatEther(aliceBalanceBefore -
↪   aliceBalanceAfter), 'ETH');
    });

    it('Malicious Buy', async () => {
        // Record initial state
        const initialMarket = await
↪   reputationMarket.getMarket(ethosUserA.profileId);
        expect(initialMarket.trustVotes).to.equal(1n);
        expect(initialMarket.distrustVotes).to.equal(1n);

        // Record Alice's balance initially
        const aliceBalanceBefore = await ethosUserA.getBalance();

        const bP = ethers.parseEther('0.01');
        let tV = 1n;
        let dV = 1n;
        let aliceInvestment;
        for(let i = 0; i < 12; i++) {
            // Alice buys trust votes
            aliceInvestment = (tV * bP) / (tV + dV);
            await alice.buyVotes({
                isPositive: true, // trust votes
                buyAmount: aliceInvestment,
                expectedVotes: 1n,  // We know this from simulation
                slippageBasisPoints: 0  // 0% slippage allowed
            });
            expect((await alice.getVotes()).trustVotes).to.equal(tV);
            tV++;

            if (i == 11) continue; // no need to manipulate further, 12 trust votes
↪   have been bought
            // Alice buys distrust votes
            aliceInvestment = (dV * bP) / (tV + dV);
```

```
            await alice.buyVotes({
                isPositive: false, // distrust votes
                buyAmount: aliceInvestment,
                expectedVotes: 1n,
                slippageBasisPoints: 0
            });
            expect((await alice.getVotes()).distrustVotes).to.equal(dV);
            dV++;
        }

        // Verify Alice's position
        const alicePosition = await alice.getVotes();
        console.log("\n\n");
        console.log("votes position (TRUST)    =", alicePosition.trustVotes);

        // Alice sells all her distrust votes
        await reputationMarket
            .connect(alice.signer)
            .sellVotes(DEFAULT.profileId, false, alicePosition.distrustVotes);
        expect((await alice.getVotes()).distrustVotes).to.equal(0);

        // Record Alice's balance now
        const aliceBalanceAfter = await ethosUserA.getBalance();

        // Log the total cost to Alice
        console.log('Cost2:', ethers.formatEther(aliceBalanceBefore -
↪   aliceBalanceAfter), 'ETH');
    });
});
```

Output:

```
  BuyVotes Cost Manipulation


votes position (TRUST)    = 12n
Cost1: 0.098439322450225045 ETH
      Normal Buy




votes position (TRUST)    = 12n
Cost2: 0.079440616542537061 ETH
      Malicious Buy (1125ms)


  2 passing (5s)
```

Add this file as `rep.bondingCurveManipulated.test.ts` inside the `ethos/packages/contracts/test/reputationMarket/` directory and run with `npm run hardhat -- test --grep "t0x1c Buy n Sell"` to see the output:

```
import { loadFixture } from '@nomicfoundation/hardhat-toolbox/network-helpers.js';
import { expect } from 'chai';
import hre from 'hardhat';
import { type ReputationMarket } from '../../typechain-types/index.js';
import { createDeployer, type EthosDeployer } from '../utils/deployEthos.js';
import { type EthosUser } from '../utils/ethosUser.js';
import { DEFAULT, MarketUser } from './utils.js';

const { ethers } = hre;

describe('t0x1c Buy n Sell', () => {
    let deployer: EthosDeployer;
    let ethosUserA: EthosUser;
    let ethosUserB: EthosUser;
    let alice: MarketUser;
    let bob: MarketUser;
    let reputationMarket: ReputationMarket;
    let market: ReputationMarket.MarketInfoStructOutput;

    beforeEach(async () => {
        deployer = await loadFixture(createDeployer);

        if (!deployer.reputationMarket.contract) {
        throw new Error('ReputationMarket contract not found');
        }
        ethosUserA = await deployer.createUser();
        await ethosUserA.setBalance('100');
        ethosUserB = await deployer.createUser();
        await ethosUserB.setBalance('100');

        alice = new MarketUser(ethosUserA.signer);
        bob = new MarketUser(ethosUserB.signer);

        reputationMarket = deployer.reputationMarket.contract;
        DEFAULT.reputationMarket = reputationMarket;
        DEFAULT.profileId = ethosUserA.profileId;
        await reputationMarket
            .connect(deployer.ADMIN)
            .setUserAllowedToCreateMarket(DEFAULT.profileId, true);
        await reputationMarket.connect(alice.signer).createMarket({ value:
↪   ethers.parseEther('0.02') });
        market = await reputationMarket.getMarket(DEFAULT.profileId);
        expect(market.profileId).to.equal(DEFAULT.profileId);
        expect(market.trustVotes).to.equal(1);
        expect(market.distrustVotes).to.equal(1);
    });
```

```
it('Buy and Sell - normal', async () => {
    // Record initial state
    const initialMarket = await
↪ reputationMarket.getMarket(ethosUserA.profileId);
    expect(initialMarket.trustVotes).to.equal(1n);
    expect(initialMarket.distrustVotes).to.equal(1n);

    // Bob buys 99 trust votes
    await bob.buyVotes({
        isPositive: true, // trust votes
        buyAmount: ethers.parseEther('0.95'),
        expectedVotes: 99n,
        slippageBasisPoints: 0  // 0% slippage allowed
    });
    expect((await bob.getVotes()).trustVotes).to.equal(99);

    const aliceBalanceBefore = await ethosUserA.getBalance();
    // Alice buys 100 trust votes
    await alice.buyVotes({
        isPositive: true, // trust votes
        buyAmount: ethers.parseEther('1.0'),
        expectedVotes: 100n,
        slippageBasisPoints: 0  // 0% slippage allowed
    });
    expect((await alice.getVotes()).trustVotes).to.equal(100);
    const aliceBalanceAfter = await ethosUserA.getBalance();

    // Log the total cost to Alice
    console.log('\n\nCost_1:', ethers.formatEther(aliceBalanceBefore -
↪ aliceBalanceAfter), 'ETH');
});

it('Buy and Sell - malicious', async () => {
    // Record initial state
    const initialMarket = await
↪ reputationMarket.getMarket(ethosUserA.profileId);
    expect(initialMarket.trustVotes).to.equal(1n);
    expect(initialMarket.distrustVotes).to.equal(1n);

    // Bob buys 99 trust votes
    await bob.buyVotes({
        isPositive: true, // trust votes
        buyAmount: ethers.parseEther('0.95'),
        expectedVotes: 99n,
        slippageBasisPoints: 0  // 0% slippage allowed
    });
    expect((await bob.getVotes()).trustVotes).to.equal(99);

    const aliceBalanceBefore = await ethosUserA.getBalance();
    // Alice first buys 99 distrust votes
```

```
        await alice.buyVotes({
            isPositive: false, // distrust votes
            buyAmount: ethers.parseEther('0.305'),
            expectedVotes: 99n,
            slippageBasisPoints: 0  // 0% slippage allowed
        });
        expect((await alice.getVotes()).distrustVotes).to.equal(99);

        // Alice then buys 100 trust votes
        await alice.buyVotes({
            isPositive: true, // trust votes
            buyAmount: ethers.parseEther('0.595'),
            expectedVotes: 100n,
            slippageBasisPoints: 0  // 0% slippage allowed
        });
        expect((await alice.getVotes()).trustVotes).to.equal(100);

        // Alice sells all her distrust votes
        await reputationMarket
            .connect(alice.signer)
            .sellVotes(DEFAULT.profileId, false, (await
↪   alice.getVotes()).distrustVotes);
        expect((await alice.getVotes()).distrustVotes).to.equal(0);

        expect((await alice.getVotes()).trustVotes).to.equal(100);

        // Record Alice's balance now
        const aliceBalanceAfter = await ethosUserA.getBalance();

        // Log the total cost to Alice
        console.log('\n\nCost_2:', ethers.formatEther(aliceBalanceBefore -
↪   aliceBalanceAfter), 'ETH');
    });
});
```

Output:

```
  t0x1c Buy n Sell


Cost_1: 0.993452851893538135 ETH
      Buy and Sell - normal (87ms)


Cost_2: 0.711616420426520863 ETH
      Buy and Sell - malicious (279ms)


  2 passing (5s)
```

Mitigation

It would be advisable to introduce a time delay between any consecutive calls to buy or sell votes by the same address. This would ensure price manipulation can't happen in the same block, thus mitigating the attack.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2214

Issue H-2: Users could overpay fees when buying votes

Source: https://github.com/sherlock-audit/2024-11-ethos-network-ii-judging/issues/314

Found by

0x486776, 0xPhantom2, 0xaxaxa, 0xc0ffEE, 0xlucky, Abhan1041, BengalCatBalu, DenTonylifer, John44, Ryonen, X12, aycozyOx, bughuntoor, chaos304, dobrevaleri, hals, irresponsible, newspacexyz, novaman33, pashap9990, smbv-1923, tmotfl, underdog, vatsal, wellbyt3, ydlee, zxriptor

Summary

The `previewFees` function in `_calculateBuy` is applied to the total `funds` being transferred by the user. This leads to users paying for more funds that they are actually transacting.

Root Cause

In ReputationMarket:960, users will specify an amount of `funds` that they are willing to pay in exchange for votes. However, the specified `funds` might not be fully used in order to buy the votes, given the following logic in `_calculateBuy`:

```
// ReputationMarket.sol

function _calculateBuy(
    Market memory market,
    bool isPositive,
    uint256 funds
  )
    private
    view
    returns (
      uint256 votesBought,
      uint256 fundsPaid,
      uint256 newVotePrice,
      uint256 protocolFee,
      uint256 donation,
      uint256 minVotePrice,
      uint256 maxVotePrice
    )
  {
    uint256 fundsAvailable;
    (fundsAvailable, protocolFee, donation) = previewFees(funds, true);
    uint256 votePrice = _calcVotePrice(market, isPositive);
```

```
    uint256 minPrice = votePrice;
    uint256 maxPrice;

    if (fundsAvailable < votePrice) {
      revert InsufficientFunds();
    }

    while (fundsAvailable >= votePrice) {
      fundsAvailable -= votePrice;
      fundsPaid += votePrice;
      votesBought++;

      market.votes[isPositive ? TRUST : DISTRUST] += 1;
      votePrice = _calcVotePrice(market, isPositive);
    }
    fundsPaid += protocolFee + donation;

    maxPrice = votePrice;

    return (votesBought, fundsPaid, votePrice, protocolFee, donation, minPrice,
↪   maxPrice);
  }
```

As shown in the snippet, the amount of votes bought is determined by a loop that will run while the `fundsAvailable` are greater than the `votePrice`. As the price of votes increases due to the buy pressure, the loop will be finished, having `fundsAvailable` (which consists of the user's submitted `funds` with the fees substracted) be nearly always greater than the actual `fundsPaid` (which consists of the price paid for each vote + fees + donation fee).

The problem with this approach is that fees are being applied to an amount that, as demonstrated, is not necessarily the total amount used to actually buy the votes.

Internal pre-conditions

None

External pre-conditions

None

Attack Path

Let's say a user 1 wants to buy 2 `TRUST` votes for a recently created market, where `basePrice` is 0,01 ETH and theres 1 `TRUST` and 1 `DISTRUST` vote already in the market.

1. The price of one vote is given by `market.votes[isPositive ? TRUST : DISTRUST] * market.basePrice) / totalVotes`, so the price is 0,005 ETH (1 * 0,01 / 2) for the first vote, and ≈ 0,0066 (2 * 0,01 / 3) for the second vote, which adds up to a total of 0,0116 ETH. A 10 % fee is applied (so fee is 0,00116), so the total that user should deposit is 0,01276.
2. At the same time, and prior to user 1 buying the votes, user 2 submits a buy transaction for one `TRUST` vote. This leaves the state of the vote prices to a different price than the expected by user 1. Still, user 1 submits the transaction with a value of 0,01276.
3. Because user 2 has triggered the buy operation prior to user 1, the initial vote price for user 1 is 0,0066 ETH (2 * 0,01 / 3). As user 1 submitted 0,01276 as `funds`, and without the 10% in fees, the `fundsAvailable` for user 1 are 0,011484. Note that 0,001276 are paid in fees.
4. While in the loop:

- First iteration: `votePrice` starts at 0,0066, and `fundsAvailable` are 0,011484, so one vote is purchased. The `fundsPaid` increases to 0,0066.
- Second iteration: `votePrice` now is at 0,0075 (3 * 0,01 / 4). `fundsAvailable` are 0,004884, so it is not enough to buy a second vote

5. The result is that user 1 has only been able to purchase a single vote. The actual transacted value has only been 0,0066 (the price of one vote), for which a 10% fee would have implied 0,00066 ETH (≈ 2,442 USD). However, as shown in step 1, user 1 has paid 0,00116 ETH (≈ 4,292 USD, **nearly the double!**).

On the long run, situations like this will arise, leading to a perpetual loss of funds for protocol users due to the increase

Impact

Medium. As shown in the "Attack Path" section, fees will be overcharged for users that buy votes. On the long term, it is easy that the total value loss exceeds 10 USD or 10% of user value, (considering that new markets might also be configured, with base prices of 0,1 or even 1 ETH).

PoC

No response

Mitigation

When triggering `buyVotes`, allow users to specify the amount of votes they want, instead of the amount of ETH to pay (similar to the logic used when selling). Then, apply the corresponding fees considering the actual amount that user will pay.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2214

Issue H-3: Market funds cannot be withdrawn because of incorrect calculation of `fundsPaid`

Source:
https://github.com/sherlock-audit/2024-11-ethos-network-ii-judging/issues/660

Found by

0rpse, 0x486776, 0xEkko, 0xPhantom2, 0xProf, 0xaxaxa, 0xgremlincat, 0xlucky, 0xmujahid002, 0xpiken, 4th05, Abhan1041, AestheticBhai, Al-Qa-qa, Artur, BengalCatBalu, Bozho, Ch_301, Cybrid, DenTonylifer, DharkArtz, HOM1T, John44, JohnTPark24, KingNFT, KlosMitSoss, Limbooo, Nave765, NickAuditor2, Pablo, Waydou, Weed0607, X0sauce, X12, ami, blutorque, bube, bughuntoor, cryptic, debugging3, dobrevaleri, farismaulana, future2_22, hals, iamnmt, kelcaM, kenzo123, mahdikarimi, nikhilx0111, novaman33, parzival, pashap9990, qandisa, rudhra1749, shui, smbv-1923, t.aksoy, tjonair, tmotfl, tobi0x18, udo, vatsal, volodya, wellbyt3, whitehair0330, y4y, ydlee, zxriptor

Summary

Funds withdrawal is blocked as fees are not deducted from fundsPaid when already being applied.

Root Cause

When votes are bought in `ReputationMarket` market, user has to pay fees to:

- donation fees going to owner of the market
- protocol fees going to treasury

This is seen in `applyFees` function below:

```
  function applyFees(
    uint256 protocolFee,
    uint256 donation,
    uint256 marketOwnerProfileId
  ) private returns (uint256 fees) {
@>    donationEscrow[donationRecipient[marketOwnerProfileId]] += donation; //
 ↪  donation fees are updated for market owner
    if (protocolFee > 0) {
@>      (bool success, ) = protocolFeeAddress.call{ value: protocolFee }(""); //
 ↪  protocolFees paid to treasury
      if (!success) revert FeeTransferFailed("Protocol fee deposit failed");
    }
```

```
    fees = protocolFee + donation;
  }
```

Next, the total amount a user pays when votes are bought is managed by the `fundsPaid` variable. The amount consists of: `cost of votes + protocol fees + donation fees`

The vulnerability exists in the execution here:

1. send protocol fees to the treasury
2. add donations to market owner's escrow
3. marketOwner is able to withdraw donations via `withdrawDonations()`

In the `buyVotes` function, protocolFee and donation are paid first as seen below:

```
applyFees(protocolFee, donation, profileId);
```

Then, when tallying the market funds, `marketFunds` is updated with `fundsPaid`. This `fundsPaid` still includes the protocolFee and donation and has not been deducted.

```
marketFunds[profileId] += fundsPaid;
```

Hence, the protocolFee and donation has been counted twice.

When a market graduates, because of the incorrect counting of `marketFunds`, the contract may not have enough funds to be withdrawn via `ReputationMarket.withdrawGraduatedMarketFunds` and results in transaction reverting.

Proof of Concept

Assume this scenario:

A market exists with 2 trust votes and 2 distrust votes, each costing 0.03 ETH. Protocol and donation fees are both set at 5%. Alice buys 2 trust votes for 0.07 ETH:

Fees (5% each): Protocol: 0.0015 ETH per vote → 0.003 ETH total. Donations: 0.0015 ETH per vote → 0.003 ETH total. Vote Cost: 0.03 ETH × 2 = 0.06 ETH. Refund: 0.07 ETH - (0.06 ETH + 0.006 ETH fees) = 0.004 ETH. The contract incorrectly records 0.066 ETH (votes + fees) as market funds.

Market owner withdraws the 0.06 ETH correctly available.

After market graduation, the contract attempts to withdraw the recorded 0.066 ETH, but only 0.06 ETH exists.

Impact:

The withdrawal fails due to insufficient funds. Funds are stuck, or other markets' funds are misallocated. If a withdrawal succeeds, it might wrongly pull ETH allocated to other markets, leading to losses for other users.

Line(s) of Code

https://github.com/sherlock-audit/2024-11-ethos-network-ii/blob/main/ethos/packages/contracts/contracts/ReputationMarket.sol#L442

https://github.com/sherlock-audit/2024-11-ethos-network-ii/blob/main/ethos/packages/contracts/contracts/ReputationMarket.sol#L920

https://github.com/sherlock-audit/2024-11-ethos-network-ii/blob/main/ethos/packages/contracts/contracts/ReputationMarket.sol#L660

https://github.com/sherlock-audit/2024-11-ethos-network-ii/blob/main/ethos/packages/contracts/contracts/ReputationMarket.sol#L1116

Recommendations

Update logic to deduct protocol fees and donations before updating `marketFunds`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2216

Issue H-4: A user can pay less in fees by vouching initially with a smaller amount and then using the `EthosVouch::increaseVouch` function to add the remaining vouch value

Source: https://github.com/sherlock-audit/2024-11-ethos-network-ii-judging/issues/714

Found by

0xKann, 0xPhantom2, Bbash, BengalCatBalu, John44, POB, Pheonix, Ryonen, SovaSlava, X12, debugging3, mahdikarimi, moray5554, shui, t0x1c, tachida2k, tjonair, zxriptor

Summary

A vulnerability in the `EthosVouch` fee mechanism allows users to reduce fees when vouching for a subject. By splitting their vouching process into multiple smaller transactions, users can partially reclaim `vouchersPoolFee`, resulting in significantly lower total fees compared to a single large transaction. This exploit undermines the intended fee structure and results in financial losses for other previous vouchers.

Root Cause

The `vouchersPoolFee` is redistributed to existing vouchers. By vouching with a smaller value initially, a user becomes an existing voucher and subsequently benefits from `vouchersPoolFee` in subsequent EthosVouch::increaseVouch calls. The logic does not distinguish between fees for new vouches and subsequent increases, enabling fee circumvention.

Internal pre-conditions

None

External pre-conditions

1. There is at least one other existing voucher to receive part of the `vouchersPoolFee`.

Attack Path

1. A user initially vouches with a smaller value (e.g., 10 ETH instead of the intended 100 ETH).
2. The user becomes an existing voucher and receives part of the `vouchersPoolFee`.
3. The user repeatedly calls `increaseVouch` in smaller increments (e.g., 10 ETH per transaction) to reach the intended total vouch value.
4. In each `increaseVouch` call, the user reclaims part of the `vouchersPoolFee`, significantly reducing the total fees paid.

Impact

1. Existing vouchers lose part of the intended fee revenue. In the provided example, the user saves approximately 2.54 ETH in fees for a 100 ETH vouch.

PoC

Example: User A wants to vouch with 100 ETH for a subject S. User B has already vouched with 1 ETH for that subject. The fees are defined as follows:

- entryProtocolFeeBasisPoints = 100 (1%)
- entryDonationFeeBasisPoints = 200 (2%)
- entryVouchersPoolFeeBasisPoints = 300 (3%)

If User A simply calls the `EthosVouch::vouchByProfileId` function with a `msg.value` of 100 ETH, they will pay approximately 0.99 ETH to the protocol, 1.96 ETH to the subject S, and 2.91 ETH to User B (the only previous voucher). This means they will pay a total of around 5.86 ETH in fees, and their vouch balance will be 94.14 ETH.

However, if User A wants to pay fewer fees, they can call the `EthosVouch::vouchByProfileId` function with a `msg.value` of 10 ETH and then call the `EthosVouch::increaseVouch` function nine more times, each with 10 ETH. By doing this, User A becomes a previous voucher and receives part of the `vouchersPoolFee`. In the end, their vouch balance will be approximately 96.68 ETH, meaning they paid 2.54 ETH less in fees (5.86 ETH - 3.32 ETH) compared to the first case.

Note: On-chain fees are excluded from the calculations, but they are much lower than the protocol fees.

Mitigation

Exclude the vouching user from receiving `vouchersPoolFee` during their own transactions

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2242

Issue M-1: authorProfileId can avoid being slashed

Source: https://github.com/sherlock-audit/2024-11-ethos-network-ii-judging/issues/1

Found by

0xAnmol, 0xPhantom2, 0xbakeng, 0xc0ffEE, BengalCatBalu, Contest-Squad, LeFy, X12, farismaulana, newspacexyz, rmdanxyz, volodya

Summary

there is not lock lock on lock on staking (and withdrawals) for the accused authorProfileId

Root Cause

According to docs their should be lock

> Any Ethos participant may act as a "whistleblower" to accuse another participant of inaccurate claims or unethical behavior. This accusation triggers a 24h lock on staking (and withdrawals) for the accused. Currently anyone can unvouch at any time

```
function unvouch(uint256 vouchId) public whenNotPaused nonReentrant {
  Vouch storage v = vouches[vouchId];
  _vouchShouldExist(vouchId);
  _vouchShouldBePossibleUnvouch(vouchId);
  // because it's $$$, you can only withdraw/unvouch to the same address you used
↪   to vouch
  // however, we don't care about the status of the address's profile; funds are
↪   always attached
  // to an address, not a profile
  if (vouches[vouchId].authorAddress != msg.sender) {
    revert AddressNotVouchAuthor(vouchId, msg.sender,
↪   vouches[vouchId].authorAddress);
  }

  v.archived = true;
  // solhint-disable-next-line not-rely-on-time
  v.activityCheckpoints.unvouchedAt = block.timestamp;
  // remove the vouch from the tracking arrays and index mappings
  _removeVouchFromArrays(v);

  // apply fees and determine how much is left to send back to the author
  (uint256 toWithdraw, ) = applyFees(v.balance, false, v.subjectProfileId);
```

```
  // set the balance to 0 and save back to storage
  v.balance = 0;
  // send the funds to the author
  // note: it sends it to the same address that vouched; not the one that called
↪  unvouch
  (bool success, ) = payable(v.authorAddress).call{ value: toWithdraw }("");
  if (!success) {
    revert FeeTransferFailed("Failed to send ETH to author");
  }

  emit Unvouched(v.vouchId, v.authorProfileId, v.subjectProfileId);
}
```

contracts/contracts/EthosVouch.sol#L452

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

Accused profile sees that a lot of complains going against him and unvouch all vouched funds before slashing

Impact

authorProfileId can avoid being slashed

PoC

No response

Mitigation

```
+    function pauseActions(uint authorProfileId) external onlyOwner{
+        ...
+    }

  function unvouch(uint256 vouchId) public whenNotPaused nonReentrant  {
```

```
+       uint256 authorProfileId = IEthosProfile(
+           contractAddressManager.getContractAddressForName(ETHOS_PROFILE)
+       ).verifiedProfileIdForAddress(msg.sender);
+     require(!isActionsPaused(authorProfileId), "actions paused")
      Vouch storage v = vouches[vouchId];
     _vouchShouldExist(vouchId);
     _vouchShouldBePossibleUnvouch(vouchId);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2284

Issue M-2: Missing slippage protection on `sellVotes()`

Source: https://github.com/sherlock-audit/2024-11-ethos-network-ii-judging/issues/451

Found by

0xAnmol, 0xDemon, 0xMosh, 0xaxaxa, 1337web3, Abhan1041, Al-Qa-qa, BengalCatBalu, Contest-Squad, DenTonylifer, HOM1T, John44, KlosMitSoss, Kyosi, LeFy, Ryonen, X12, blutorque, bughuntoor, cryptic, dobrevaleri, durov, iamthesvn, immeas, justAWanderKid, ke1caM, mahdikarimi, nikhilx0111, pashap9990, qandisa, redbeans, rscodes, smbv-1923, t.aksoy, t0x1c, tmotfl, underdog, zeGarcao, zhenyazhd, zxriptor

Summary

The `ReputationMarket` contract provides preview functions (simulateBuy() and simulateSell()) to estimate outcomes before actual transactions (buyVotes() and sellVotes()). While buyVotes() includes slippage protection against price changes between simulation and execution, sellVotes() lacks this safeguard. While Base L2's private mempool prevents traditional frontrunning, users are still exposed to two risks:

1. Market volatility between simulation and execution could result in receiving fewer funds than expected
2. The sequencer prioritizes transactions with higher fees (ref), allowing users paying higher fees to execute trades first, potentially leading to unfavorable price movements for pending transactions with lower fees.

Root Cause

sellVotes() is missing slippage protection.

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. User calls `simulateSell()` to preview expected returns
2. Market experiences high sell volume, causing price decline
3. User submits `sellVotes()` transaction with outdated price expectations
4. Due to missing slippage protection, transaction executes at significantly lower price than simulated, resulting in unexpected losses

Impact

Loss of assets for the affected users.

PoC

No response

Mitigation

Implement a slippage control that allows the users to revert if the amount they received is less than the amount they expected.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2214

Issue M-3: Separate calculation of fees in applyFees results in inflated total fee percentage.

Source: https://github.com/sherlock-audit/2024-11-ethos-network-ii-judging/issues/637

Found by

0xMosh, 0xProf, 0xaxaxa, 0xpiken, DigiSafe, HOM1T, JohnTPark24, ami, copperscrewer, debugging3, dobrevaleri, eLSeR17, future2_22, newspacexyz, pashap9990, tobi0x18, whitehair0330, zhenyazhd

Summary

The separate calculation of fees in applyFees using multiple calls to calcFee causes a higher total fee percentage than expected. This leads to users being overcharged because each fee is calculated independently, rather than as a proportion of the total deposit. As a result, the total effective fee exceeds the intended basis points when multiple fees are applied.

Root Cause

In thosVouch.sol:936 : https://github.com/sherlock-audit/2024-11-ethos-network-ii/blob/main/ethos/packages/contracts/contracts/EthosVouch.sol#L936

In the applyFees function of the EthosVouch contract, the protocol fee, donation fee, and vouchers pool fee are calculated independently using the calcFee function. This results in compounding effects where the combined fees are higher than intended.

Internal pre-conditions

1. applyFees is called with amount and configured basis points for protocol, donation, and vouchers pool fees.
2. Each fee (protocol, donation, and vouchers pool) is calculated using calcFee, which computes the fee based on the provided basis points independently of other fees.

External pre-conditions

1. A user initiates an operation that triggers applyFees, such as buying or selling votes, where multiple fees are involved.

2. The basis points for each fee (entryProtocolFeeBasisPoints, entryDonationFeeBasisPoints, entryVouchersPoolFeeBasisPoints) are configured with non-zero values.

Attack Path

1. The user makes a transaction that triggers applyFees (e.g., buying votes).
2. The protocol calculates multiple fees (protocol, donation, vouchers pool) using calcFee independently.
3. Due to separate fee calculations, the total effective fee percentage exceeds the sum of the intended basis points, leading to user overcharging.

Impact

Affected users are overcharged due to inflated fees. For example: • If entryProtocolFeeBasisPoints = 2000 (20%) and entryDonationFeeBasisPoints = 2500 (25%), the total fee is expected to be 4500 basis points (45%). However, due to independent calculations, the effective fee becomes approximately 5789 basis points (57.89%), as shown in the testFee example.

The protocol's reputation and user trust may suffer due to higher-than-expected charges.

PoC

```
function testFee() external {
    uint256 feeBasisPoints = 2000; // 20%
    uint256 feeBasisPoints2 = 2500; // 25%
    uint256 total = 300000000000000; // 0.3 ETH

    // Separate fee calculation
    uint256 fee1 = total -
        (total.mulDiv(10000, (10000 + feeBasisPoints), Math.Rounding.Floor));
    uint256 fee2 = total -
        (total.mulDiv(10000, (10000 + feeBasisPoints2), Math.Rounding.Floor));

    console.log(fee1 + fee2, total - fee1 - fee2); // Inflated fee: 5789 basis
↪   points (~57.89%)

    // Combined fee calculation
    uint256 fee3 = total -
        (total.mulDiv(10000, (10000 + feeBasisPoints2 + feeBasisPoints),
↪   Math.Rounding.Floor));
```

```
    console.log(fee3, total - fee3); // Correct fee: 4500 basis points (45%)
}
```

Mitigation

1. Calculate the total fee for all basis points in one step.
2. Distribute the total fee proportionally to the respective components.
3. Update applyFees to use the combined fee calculation logic, ensuring the total effective fee matches the expected sum of basis points.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/trust-ethos/ethos/pull/2243

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.