VotingBooth Security Audit Report by MK (11th DEC 2024)

High Risk Findings

[H-1] Miscalculation in `rewardPerVoter`

Description:

The `_distributeRewards()` function contains a miscalculation in the `rewardPerVoter` logic, leading to reduced rewards being distributed to voters. Additionally, a significant leftover amount remains locked in the contract, which cannot be withdrawn or transferred.

- Suppose s_totalAllowedVoters = 5
- Votes needed to reach quorum = 3 votes (Since 3 * 100\5 >= 51)
- Let us have totalRewards = 10 ether
- 1st voter votes AGAINST the proposal.
- 2nd voter votes FOR the proposal.
- 3rd voter votes FOR the proposal.
- Hence, totalVotes = 3 and totalVotesFor = 2.
- rewardPerVoter is calculated as per L192 which says: rewardPerVoter = totalRewards\ totalVotes => rewardPerVoter = 10 ether \ 3. This equals 3333333333333333 wei. This is incorrect & should be: rewardPerVoter = totalRewards \ totalVotesFor => rewardPerVoter = 10 ether \ 2. Which equals 5000000000000000000 wei. Also, the current reward distribution looks like this: 2nd voter receives 3333333333333333 wei 3rd voter receives 3333333333333334 wei Dust left in the contract = 10 ether - 3333333333333333 wei - 3333333333333334 wei = 3333333333333333 wei.

Impact:

The incorrect calculation of rewardPerVoter has significant implications for the contract's functionality and fairness. Voters receive lower rewards than intended, undermining the trust and incentives for participation. Furthermore, the leftover funds locked in the contract represent an inefficiency that cannot be rectified due to the inability to withdraw or transfer these funds. This issue could erode user confidence in the system and potentially lead to a loss of engagement, highlighting the critical need for accurate calculations and proper fund management in smart contracts.

Proof of Concept:

1. Write the Unit test:

```
function testIncorrectReward() public {
vm.prank(address(0x1));
booth.vote(false);
vm.prank(address(0x2));
booth.vote(true);
```

```
vm.prank(address(0x3));
booth.vote(true);

assert(!booth.isActive());
assertEq(address(0x2).balance, 5_000_000_000_000_000_000); //@audit :should have
been 5000000000000000000 wei = 5 ether
assertEq(address(0x3).balance, 5_000_000_000_000_000_000); //@audit :should have
been 5000000000000000000 wei = 5 ether
assertEq(address(booth).balance, 0, "dust remains"); //@audit : should have been
0, but is 3333333333333333333
}
```

2. Output:

```
Compiler run successful!

Ran 1 test for test/VotingBoothTest.t.sol:VotingBoothTest
[FAIL] testIncorrectReward() (gas: 281201)
Logs:
  Error: a == b not satisfied [uint]
        Left: 3333333333333333333
       Right: 5000000000000000000
  Error: a == b not satisfied [uint]
        Left: 3333333333333333334
       Right: 5000000000000000000
  Error: dust remains
  Error: a == b not satisfied [uint]
        Left: 3333333333333333333
        Left: 3333333333333333334
       Right: 5000000000000000000
  Error: dust remains
  Error: a == b not satisfied [uint]
        Left: 3333333333333333333
       Right: 5000000000000000000
  Error: dust remains
  Error: a == b not satisfied [uint]
        Left: 3333333333333333333
  Error: dust remains
  Error: a == b not satisfied [uint]
        Left: 3333333333333333333
  Error: a == b not satisfied [uint]
        Left: 3333333333333333333
        Left: 3333333333333333333
       Right: 0

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.60ms (1.41ms
CPU time)

Ran 1 test suite in 30.87ms (8.60ms CPU time): 0 tests passed, 1 failed, 0 skipped
(1 total tests)
```

Recommended Mitigation:

1. Correct the Calculation Logic:

Modify the `_distributeRewards()` function to ensure that `rewardPerVoter` is calculated based on `totalVotesFor` rather than `totalVotes`. The corrected formula should be:

```
rewardPerVoter = totalRewards / totalVotesFor;
```

This ensures that rewards are distributed only to voters who voted for the proposal, preventing the under-distribution of rewards.

2. Implement Dust Handling:

Implement a mechanism to handle leftover funds (dust) in the contract. You can add a `withdraw()` or `transfer()` function that allows the remaining funds to be returned to the contract owner or redistributed appropriately.

```
function withdrawDust() external onlyOwner {
    payable(owner()).transfer(address(this).balance);
}
```

Medium Risk Findings

[M-1] Unsupported PUSH0 Opcode in Solidity 0.8.23 for Arbitrum Chain Deployment

Description:

The contract, compiled with Solidity 0.8.23, uses the unsupported PUSH0 opcode on the Arbitrum chain, preventing deployment. Solidity versions above 0.8.19 may cause deployment failures or improper functioning due to this incompatibility, leading to potential issues.

Impact:

The use of the unsupported PUSH0 opcode in Solidity 0.8.23 prevents the contract from being deployed on the Arbitrum chain. This issue could lead to deployment failures or malfunctioning contracts, undermining the contract's functionality and causing disruption in the intended operations. It is critical to ensure compatibility with all target networks to avoid such deployment issues.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;
```

Recommended Mitigation:

1. To mitigate this issue, update the contract to use a compatible Solidity version (0.8.19 or below) that does not include the unsupported PUSH0 opcode. Recompile the contract with a supported version to ensure proper deployment and functionality on the Arbitrum chain. Additionally, test the contract on the target network to confirm compatibility before deployment.

Low Risk Findings

[L-1] Leftover Dust Can Persist After _distributeRewards()

Description:

The _distributeRewards() function attempts to eliminate dust by rounding up the reward for the last voter by 1 wei. However, this approach is insufficient, as the dust could exceed 1 wei. The leftover amount becomes irretrievable, as the contract lacks a withdrawal, transfer, or selfDestruct function. The assumption that rounding up always resolves dust is incorrect for various combinations of totalRewards and totalVotes.

Impact:

The failure to properly handle dust in _distributeRewards() results in irretrievable leftover funds, leading to inefficiencies in the contract. These unclaimed amounts could accumulate over time, wasting resources and undermining the protocol's effectiveness. The lack of a mechanism to withdraw or transfer these funds further exacerbates the issue.

Proof of Concept:

1. Make the following modification in `test/VotingBoothTest.t.sol` and then run `forge test --mt testVotePassesAndMoneyIsSent -vv.` The assertion in the existing `testVotePassesAndMoneyIsSent()` test will fail because 1 wei of dust remains after reward distribution. This occurs because `20e18 % 3 = 2 wei`, not `1 wei.`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import {VotingBooth} from "../src/VotingBooth.sol";
import {Test} from "forge-std/Test.sol";
import {_CheatCodes} from "./mocks/CheatCodes.t.sol";

contract VotingBoothTest is Test {
    // eth reward
    // uint256 constant ETH_REWARD = 10e18; @removed this line.
       uint256 constant ETH_REWARD = 20e18; @modified into this.
```

2. Output:

```
Compiler run successful!

Ran 1 test for test/VotingBoothTest.t.sol:VotingBoothTest
[FAIL: panic: assertion failed (0x01)] testVotePassesAndMoneyIsSent() (gas:
268551)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.50ms (902.21µs
CPU time)

Ran 1 test suite in 121.98ms (8.50ms CPU time): 0 tests passed, 1 failed, 0
skipped (1 total tests)
```

Recommended Mitigation:

1. Adjust the `_distributeRewards()` function to redistribute or return leftover rewards to the contract balance.
2. Implement a function to allow the contract owner to withdraw or transfer unclaimed rewards, including dust