GoatTech Audit

Version 1.0

killua

GoatTech Audit Report

killua

April 9th, 2024

Prepared by: killua

Table of Contents

Table of Contents

Protocol Summary

- Goat.Tech is a social-financial game, where users play by mainly staking ETH in each other's "Trust Pool" to earn 10 types of rewards, increase reputation (Trust Score), and find out who's the GOAT (highest Trust Score). Trust Score is fully on-chain; can be used to attract, assess, and target Web3 prospects. Who needs? KOLs, founders, investors, and more.

The contracts will be deployed on Arbitrum One. While we make sure that even devteam cannot touch users' locked funds in the Locker contracts, we maintain a certain level of centralization in order to intervene when there are bugs or exploits or urgent needs to upgrade contracts. The Controller contract contains core logic and can be upgraded. Despite a certain level of centralization, it's impossible for the development team to access users' locked funds in Locker contracts.

There are 2 roles - owner and admin. All contracts have the same owner address. Owner can change admin addresses. In our case, admin addresses are internal contract addresses, not EOAs, so that only internal contracts can call each other. The natural process of software stability takes time and iteration. We're committed to removing upgradability from our smart contracts, but this process must first run its course.

Disclaimer

- The killua team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

Scope

- Code Base: https://github.com/cantina-forks/goattechlabs-smart-contracts
- approx nSLOC= 4000
- In Scope:

```
  #-- Contracts
|   #-- Controller.sol
|   #-- DCT.sol
|   #-- GlobalAccessControl.sol
|   #-- PoolFactory.sol
    #--PrivateVester.sol
    #--Voting.sol
```

```
    #--EthSharing.sol
    #-- modules
      #--AccessControl.sol
      #--Cashier.sol
      #--DToken.sol
      #--Distribution.sol
      #--Earning.sol
      #--Initializable.sol
      #--Locker.sol
      #--PERC20.sol
      #--UserAccessControl.sol
      #--Vester.sol
```

Issues found

Severity	Number of issues found
High	1
Medium	1
Low	0
Info	0
Gas Optimizations	0
Total	2

Findings

[H-01] Potential Double Claiming of rewards by Winner(Attacker/Defender) Post-Vote Finalization

Description - `Voting:_tryFinalize` function not only determines the winner of the vote, but also transfers the reward to the winner (either attacker or defender) based on the vote outcome. Voting:_tryFinalize function does not set isClaimed flag to true after transferring rewards to the winner. The absence of such a flag means that after the _tryFinalize function executes and distributes rewards to the winning party, nothing technically prevents the winner (either attacker or defender) from invoking the claimFor function to claim rewards again.

```
if (vote.isAttackerWon) {
```

```
            vote.winVal = LPercentage.getPercentA(vote.dEthValue, vote.
                voterPercent);
            vote.winnerPower = vote.attackerPower;
            uint toWinnerVal = vote.aEthValue + vote.dEthValue - vote.
                winVal;
            address payable to = payable(vote.attacker);
            // refund to attacker
            LLido.sellWsteth(toWinnerVal);
            LLido.wethToEth();
            to.send(address(this).balance);
            clean();
        } else {
            vote.winVal = LPercentage.getPercentA(vote.aEthValue, vote.
                voterPercent);
            vote.winnerPower = vote.defenderPower;
            uint unfreezeVal = vote.dEthValue;
            uint rewardWinnerVal = vote.aEthValue - vote.winVal;
            address payable to = payable(vote.defender);

            _eEarning.clean();
            // refund to defender
            _cashOut(address(_eEarning), unfreezeVal);
            _eEarning.update(to, false);
            //reward to defender
            _cashOut(address(_eEarning), rewardWinnerVal);
            _eEarning.update(to, true);
        }
```

Impact - This could lead to a scenario where a winner unfairly claims more than their entitled reward amount, potentially draining resources intended for other participants and destabilizing the reward distribution system.

Likelihood - If the contract is used as intended, and winners are aware of this oversight, there's a high likelihood they could exploit this bug.

Recommendation - Add in `_tryFinalise` function

```

++ vote.isClaimed[to] = true
```

[M-01] [Logical Error]:Discrepancy in Attacker Win Condition check Against Quorum Requirement

Description - Voting::_tryFinalize function is responsible for determining the outcome.It uses a strict greater-than (>) comparison to determine if the attacker has won the vote. - In vote.isAttackerWon = vote.attackerPower > reqPower; reqPower represents the Quorum required in the vote. - However, as per documentation, an attacker wins if their power is greater than or equal to (>=) the required quorum.

- If left uncorrected, this discrepancy could lead to situations where an attacker who meets exactly the quorum does not win the vote as they should per protocol rules.

- As per Documentation : If Yae side wins (% Yes vote >= Yes_quorum) Docs url: https://academy.goat.tech/goat.tech/dapp/reputation-challenge/attack-defense-voting-reward

Recommendation - This change ensures that when an attacker's power equals the required quorum, they are correctly recognized as having won the vote.

- Before fix:

```
vote.isAttackerWon = vote.attackerPower > reqPower;
```

- After fix:

```
vote.isAttackerWon = vote.attackerPower >= reqPower;
```