Pashov Audit Group

BOB Staking Security Review

October 18th 2025 - October 21st 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About BOB Staking

BOB Staking is a staking system that implements an unbonding mechanism and instant withdrawal features. Users can stake tokens, earn rewards, and choose between a standard unbonding process or instant withdrawal with a penalty.

5. Executive Summary

A time-boxed security review of the **bob-collective/bob-staking** repository was done by Pashov Audit Group, during which **zark**, **Tejas Warambhe**, **IvanFitro**, **afriauditor** engaged to review **BOB Staking**. A total of **20** issues were uncovered.

Protocol Summary	
Project Name	BOB Staking
Protocol Type	Token Staking
Timeline	October 18th 2025 - October 21st 2025

Review commit hash:
- 73158d3c424bdc7c0402bf59c6e42f4dd261f9d6 (bob-collective/bob-staking)

Fixes review commit hash:
- 44842c634a3a4b87b889384de2f648c8e4fab1b8 (bob-collective/bob-staking)

Scope

`BobStaking.sol` `BonusWrapper.sol`

6. Findings

Findings count

Severity	Amount
Critical	2
High	2
Medium	3
Low	13
Total findings	**20**

Summary of findings

ID	Title	Severity	Status
[C-01]	`instantWithdraw()` does not transfer `_amountForContract`, locking tokens	Critical	Resolved
[C-02]	Stakes not forwarded post-delegation, positions unwithdrawable	Critical	Resolved
[H-01]	Bonuses obtainable without proper locking due to flawed lock period	High	Resolved
[H-02]	Delegating to `address(0)` empties contract via `alterGovernanceDelegatee()`	High	Resolved
[M-01]	Instant withdraw lets users self-fund residuals with their own penalty	Medium	Acknowledged
[M-02]	Condition setter functions are broken	Medium	Resolved
[M-03]	DoS of staking due to unguarded receiver lock period	Medium	Resolved
[L-01]	Residual claim reverts on shortage	Low	Acknowledged
[L-02]	Expired lock accepts new stake	Low	Acknowledged
[L-03]	Missing validation allows `bonusEndTime` to be set to past timestamps	Low	Resolved
[L-04]	Missing events on key setters	Low	Acknowledged
[L-05]	Multistep division leads to loss of precision	Low	Acknowledged

ID	Title	Severity	Status
[L-06]	`withdrawRewardTokens()` allows excess withdrawal ignoring `residualRewardBalance`	Low	Acknowledged
[L-07]	`TIME_UNIT` is not exactly one year	Low	**Resolved**
[L-08]	Users cannot opt out of hybrid node delegation	Low	**Resolved**
[L-09]	Unguarded `claimRewards()` can be leveraged to deny Instant withdrawal fees	Low	**Resolved**
[L-10]	In `_setStakingCondition()` sums can exceed `REWARD_RATIO_DENOMINATOR`	Low	Acknowledged
[L-11]	Residual recycling lets attackers farm wrapper bonus without new capital	Low	**Resolved**
[L-12]	Anyone can trigger others' claims causing front-running reward loss	Low	**Resolved**
[L-13]	Boost window mis-scaled dividing by `TIME_UNIT` not 30 days	Low	Acknowledged

Critical findings

[C-01] `instantWithdraw()` does not transfer `_amountForContract`, locking tokens

Severity

Impact: High

Likelihood: High

Description

`instantWithdraw()` is used to immediately withdraw all staked tokens, applying a penalty for the early withdrawal.

```
function instantWithdraw(address _receiver) external nonReentrant {
        if (stakers[_stakeMsgSender()].unlockTimestamp > block.timestamp) {
            revert TokensLocked();
        }

        if (unbondEndTimes[_stakeMsgSender()] != 0) {
            revert UnbondAlreadyStarted();
        }

        _claimRewards(_stakeMsgSender(), false);

        uint256 amount = stakers[_stakeMsgSender()].amountStaked;
        if (amount == 0) revert NotEnoughBalance();

        stakingTokenBalance -= amount;

        // Calculate the penalty amount to the user
        uint256 _amountForUser = (amount * instantWithdrawalRate) / 100;
        uint256 _amountForContract = amount - _amountForUser;

        rewardTokenBalance += _amountForContract;

@>      if (stakers[_stakeMsgSender()].governanceDelegatee != address(0)) {
            // If the user has a governance delegatee, the tokens are stored in the surrogate
contract
            DelegationSurrogate surrogate =
storedSurrogates[stakers[_stakeMsgSender()].governanceDelegatee];
            IERC20(stakingToken).safeTransferFrom(address(surrogate), _receiver,
_amountForUser);
        } else {
            // If the user does not have a governance delegatee, the tokens are stored in this
contract
            IERC20(stakingToken).safeTransfer(_receiver, _amountForUser);
        }
```

```
        delete stakers[_stakeMsgSender()];

        emit TokensWithdrawn(_stakeMsgSender(), _receiver, amount);
    }
```

If a user delegates their tokens, the `_amountForUser` is transferred from the surrogate to the receiver. The issue is that `_amountForContract` is not transferred from the surrogate to the `BobStaking` contract, causing those tokens to become permanently stuck in the surrogate.

Moreover, the penalty is added to `rewardTokenBalance` but never actually transferred to the contract, which causes `rewardTokenBalance` to reflect a higher value than the contract truly holds.

To better illustrate the issue, copy the following POC into `BobStaking.t.sol`.

For the test to run properly, set the `storedSurrogates` mapping to `public` and add the following import to the test file: `import {DelegationSurrogate} from "@tally/staker/DelegationSurrogate.sol"`.

```
function test_PenaltyIsNotTransferedToBobStakingContracts() public {
        // Deposit reward tokens
        stakingToken.approve(address(stakeContract), 1000 ether);
        stakeContract.depositRewardTokens(1000 ether);

        uint256 stakedAmount = 1 ether;
        vm.prank(stakerOne);
        stakeContract.stake(stakedAmount, stakerOne, 0);

        address delegatee = makeAddr("delegatee one");
        vm.prank(address(this));
        address[] memory whitelistedGovernanceDelegateesToAdd = new address[](1);
        whitelistedGovernanceDelegateesToAdd[0] = delegatee;
        stakeContract.setWhitelistedDelegatees(
            whitelistedGovernanceDelegateesToAdd, new address[](0), new address[](0), new
address[](0)
        );

        uint256 timeOfStake = vm.getBlockTimestamp();

        vm.prank(stakerOne);
        stakeContract.alterGovernanceDelegatee(delegatee);

        assertEq(stakingToken.getVotes(delegatee), stakedAmount);

        uint256 contractBalanceBefore = stakingToken.balanceOf(address(stakeContract));

        vm.prank(stakerOne);
        stakeContract.instantWithdraw(stakerOne);

        uint256 contractBalanceAfter = stakingToken.balanceOf(address(stakeContract));

        DelegationSurrogate surrogate = stakeContract.storedSurrogates(delegatee);

        uint256 surrogateBalance = stakingToken.balanceOf(address(surrogate));
```

```
        assertEq(contractBalanceBefore, contractBalanceAfter);
        assertEq(surrogateBalance, stakedAmount / 2);
    }
```

Recommendations

To solve the issue, transfer `_amountForContract` to the `BobStaking` contract.

```
function instantWithdraw(address _receiver) external nonReentrant {
        if (stakers[_stakeMsgSender()].unlockTimestamp > block.timestamp) {
            revert TokensLocked();
        }

        if (unbondEndTimes[_stakeMsgSender()] != 0) {
            revert UnbondAlreadyStarted();
        }

        _claimRewards(_stakeMsgSender(), false);

        uint256 amount = stakers[_stakeMsgSender()].amountStaked;
        if (amount == 0) revert NotEnoughBalance();

        stakingTokenBalance -= amount;

        // Calculate the penalty amount to the user
        uint256 _amountForUser = (amount * instantWithdrawalRate) / 100;
        uint256 _amountForContract = amount - _amountForUser;

        rewardTokenBalance += _amountForContract;

        if (stakers[_stakeMsgSender()].governanceDelegatee != address(0)) {
            // If the user has a governance delegatee, the tokens are stored in the surrogate
contract
            DelegationSurrogate surrogate =
storedSurrogates[stakers[_stakeMsgSender()].governanceDelegatee];
            IERC20(stakingToken).safeTransferFrom(address(surrogate), _receiver,
_amountForUser);
+           IERC20(stakingToken).safeTransferFrom(address(surrogate), address(this),
_amountForContract);
        } else {
            // If the user does not have a governance delegatee, the tokens are stored in this
contract
            IERC20(stakingToken).safeTransfer(_receiver, _amountForUser);
        }

        delete stakers[_stakeMsgSender()];

        emit TokensWithdrawn(_stakeMsgSender(), _receiver, amount);
    }
```

[C-02] Stakes not forwarded post-delegation, positions unwithdrawable

Severity

Impact: High

Likelihood: High

Description

In `BobStaking`, once a user delegates governance via `alterGovernanceDelegatee`, their existing stake is moved to a `DelegationSurrogate`. However, later calls to `stake(_amount, receiver, lockPeriod)` **keep new tokens in the staking contract**:

```
IERC20(_stakingToken).safeTransferFrom(_stakeMsgSender(), address(this), _amount);
stakers[receiver].amountStaked += _amount;
```

No forwarding occurs when `stakers[receiver].governanceDelegatee != address(0)`. Exit paths then **assume all** `amountStaked` sits in the surrogate:

- `unbond()` tries `safeTransferFrom(surrogate, this, amountStaked);`
- `instantWithdraw()` tries `safeTransferFrom(surrogate, _receiver, _amountForUser);`

If part of the stake stayed in this contract (common after re-staking), the surrogate **doesn't hold enough** (and hasn't approved), so these calls **revert**. The user cannot unbond or instant-withdraw → funds are effectively stuck.

Minimal repro

1. Stake 100 → delegate → 100 moved to surrogate.
2. Stake 50 again → 50 remains in staking contract; `amountStaked = 150`.
3. Call `unbond()` or `instantWithdraw()` → contract tries to pull 150 from surrogate → **revert**.

Recommendations

- **Enforce a single custody location when delegated (preferred):** In `stake()`, if `governanceDelegatee != 0`, immediately forward `_amount` to the user's surrogate:

```
solidity if (stakers[receiver].governanceDelegatee != address(0))
{ DelegationSurrogate s =
storedSurrogates[stakers[receiver].governanceDelegatee];
IERC20(stakingToken).safeTransfer(address(s), _amount); }
```

High findings

[H-01] Bonuses obtainable without proper locking due to flawed lock period

Severity

Impact: High

Likelihood: Medium

Description

`stake()` in BonusWrapper allows users to lock their tokens for a specified duration in exchange for a staking bonus.

```
function stake(uint256 amount, address receiver, uint80 lockPeriod) external nonReentrant {
        // If the bonus period has ended, the lock period must be 0
        if (lockPeriod != 0 && bonusEndTime < block.timestamp) {
            revert BonusPeriodEnded();
        }

        // First transfer the user's tokens to this contract
        uint256 balanceBefore = IERC20(stakingToken).balanceOf(address(this));
        IERC20(stakingToken).safeTransferFrom(msg.sender, address(this), amount);
        uint256 actualAmount = IERC20(stakingToken).balanceOf(address(this)) - balanceBefore;

        uint256 bonus = _calculateBonus(actualAmount, lockPeriod);
        uint256 totalAmount;
        if (bonus > 0) {
            // try to claim the bonus for the user from the reward owner, revert if it fails.
Reward owner needs to top up their account
            IERC20(stakingToken).safeTransferFrom(rewardOwner, address(this), bonus);
            totalAmount = amount + bonus;
            emit TokensBonus(receiver, bonus);
        } else {
            totalAmount = amount;
        }

        //staking is BobStaking
        stakingToken.approve(address(staking), totalAmount);
        staking.stake(totalAmount, receiver, lockPeriod);
    }
````

This invokes the `stake()` in BobStaking.

```solidity
function stake(uint256 _amount, address receiver, uint80 lockPeriod) external nonReentrant {
        if (_amount == 0) revert ZeroTokenStake();
        // lock period must be valid
        //this needs to be sync with the lockPeriods in BonusWrapper
```

```
        if (!_contains(lockPeriods, lockPeriod)) {
            revert InvalidLockPeriod();
        }
        // If the user already has a lock period, the lock period supplied must be the same as
the existing lock period
        if (stakers[receiver].lockPeriod != 0 && stakers[receiver].lockPeriod != lockPeriod) {
            revert InconsistentLockPeriod();
        }
        if (unbondEndTimes[receiver] != 0) {
            revert UnbondAlreadyStarted();
        }

        address _stakingToken = stakingToken;

        if (stakers[receiver].amountStaked > 0) {
            _updateUnclaimedRewardsForStaker(receiver);
        } else {
            stakers[receiver].timeOfLastUpdate = uint80(block.timestamp);
            stakers[receiver].conditionIdOflastUpdate = nextConditionId - 1;
            stakers[receiver].lockPeriod = lockPeriod;
            stakers[receiver].unlockTimestamp = uint80(block.timestamp) + lockPeriod;
        }

        IERC20(_stakingToken).safeTransferFrom(_stakeMsgSender(), address(this), _amount);

        stakers[receiver].amountStaked += _amount;
        stakingTokenBalance += _amount;

        emit TokensStaked(receiver, _amount);
    }
```

If it's the first time the user stakes, their information is stored in the `stakers` mapping. For subsequent stakes, the provided `lockPeriod` must match the existing one. However, this condition can be bypassed if the user first stakes with `lockPeriod = 0` and later stakes with a different `lockPeriod`.

```
if (stakers[receiver].lockPeriod != 0 && stakers[receiver].lockPeriod != lockPeriod) {
            revert InconsistentLockPeriod();
        }
```

A user can first stake with `lockPeriod = 0` and later stake again with a different `lockPeriod` (e.g. `21 * 30 days`). The contract accepts the new `lockPeriod` because `stakers[receiver].lockPeriod == 0`, the check never triggers, and the function does not revert.

Since `unlockTimestamp` is only set on the first stake, users will use the initial `lockPeriod` but still receive the bonus for locking, even though they are not actually locking their tokens, as `unlockTimestamp` is only initialized during the first stake.

An attacker can exploit this by first staking with `lockPeriod = 0`, then staking again with a long lock period to obtain the maximum bonus without actually locking their tokens.

They can still withdraw immediately via `instantWithdraw()` (subject only to the penalty), effectively gaining the bonus while avoiding the intended lock. Alternatively, they can call `unbond()` and wait for the unbonding period to withdraw, achieving the same advantage without ever truly locking their tokens.

Additionally, this flaw also allows other timing-related exploits observed in similar staking scenarios:

- A user can stake a minimal amount with the shortest lock period (e.g., 3 months) and near the end of this lock, stake a large amount for the same period. Since the unlock time is not updated, they can withdraw almost immediately while still receiving a large bonus.
- A user can stake a dust amount for a very long lock (e.g., 21 months), wait for it to expire, then stake a large amount again with the same `lockPeriod` and instantly withdraw, draining the `rewardOwner`.
- If the admin later uses `BonusWrapper::setBonusEndTime()`, attackers can deliberately keep small stakes active to remain eligible and time their large stakes to maximize bonuses unfairly.

To reproduce the main issue, copy the following POC into `BonusWrapper.t.sol`.

```
function test_FreeBonusWithoutLockingTokens() public {
        vm.startPrank(staker);

        stakingToken.approve(address(bonusWrapper), type(uint256).max);

        //lockPeriod = 0
        bonusWrapper.stake(400 * 10 ** 18, staker, 0);

        //lockPeriod = 21 * 30 days
        vm.expectEmit();
        emit BonusWrapper.TokensBonus(staker, 800 * 10 ** 18);
        bonusWrapper.stake(400 * 10 ** 18, staker, 21 * 30 days);

        stakeContract.unbond();
    }
```

Recommendations

To solve the problem, check if the user has `amountStaked > 0`, and if `lockPeriod == 0`, prevent setting a different `lockPeriod`.

```
if (stakers[receiver].lockPeriod != 0 && stakers[receiver].lockPeriod != lockPeriod ||
stakers[receiver].amountStaked > 0 &&  stakers[receiver].lockPeriod != lockPeriod) {
            revert InconsistentLockPeriod();
        }
```

Additionally, ensure that when an existing staker deposits more with a non-zero `lockPeriod`, the `unlockTimestamp` is updated or extended to maintain a valid locking period for all staked tokens. Optionally, a short grace period could be introduced for legitimate users to add to existing stakes without creating timing advantages.

[H-02] Delegating to `address(0)` empties contract via `alterGovernanceDelegatee()`

Severity

Impact: High

Likelihood: Medium

Description

`alterGovernanceDelegatee()` is used to delegate a user's tokens to a delegatee.

```
function alterGovernanceDelegatee(address newDelegatee) external nonReentrant {
        Staker storage staker = stakers[_stakeMsgSender()];
        if (staker.governanceDelegatee == newDelegatee) revert DelegateeUnchanged();
        if (staker.amountStaked == 0) revert ZeroTokenStake();

        //update rewards before
        _updateUnclaimedRewardsForStaker(_stakeMsgSender());

        DelegationSurrogate newSurrogate = _fetchOrDeploySurrogate(newDelegatee);

        if (staker.governanceDelegatee == address(0)) {
            // First time delegation, staker's tokens are in this contract
            IERC20(stakingToken).safeTransfer(address(newSurrogate), staker.amountStaked);
        } else {
            // Changing delegation, staker's tokens are in the old surrogate
            DelegationSurrogate oldSurrogate = storedSurrogates[staker.governanceDelegatee];
            IERC20(stakingToken).safeTransferFrom(address(oldSurrogate), address(newSurrogate),
staker.amountStaked);
        }

        staker.amountStaked = staker.amountStaked;
        staker.governanceDelegatee = newDelegatee;

        emit GovernanceDelegateeAltered(_stakeMsgSender(), newDelegatee);
    }
```

`alterGovernanceDelegatee()` delegates a user's staked tokens to a delegatee. On a user's first delegation, tokens are transferred from the contract to a surrogate; on subsequent delegations, tokens are moved between surrogates.

A user who previously delegated can set `newDelegatee = address(0)`. When that happens, the next transfer pulls tokens from the contract balance (which holds other users' stakes and rewards) rather than from the user's own staked amount.

By repeatedly delegating first to a non-zero address and then to `address(0)`, an attacker can progressively drain the contract balance by delegating tokens to the zero address and transferring the tokens to the surrogate.

To reproduce the issue, copy the following POC into `BobStaking.t.sol`.

```
function test_ContractCanBeCompletelyEmptied() public {
       // Deposit reward tokens
        stakingToken.approve(address(stakeContract), 1000 ether);
        stakeContract.depositRewardTokens(1000 ether);

        uint256 stakedAmount = 1 ether;
        vm.prank(stakerOne);
        stakeContract.stake(stakedAmount, stakerOne, 0);

        address delegatee = makeAddr("delegatee one");
        vm.prank(address(this));
        address[] memory whitelistedGovernanceDelegateesToAdd = new address[](1);
        whitelistedGovernanceDelegateesToAdd[0] = delegatee;
        stakeContract.setWhitelistedDelegatees(
            whitelistedGovernanceDelegateesToAdd, new address[](0), new address[](0), new
address[](0)
        );

        uint256 timeOfStake = vm.getBlockTimestamp();

        address delegate0 = address(0);

        uint256 contractBalanceBefore = stakingToken.balanceOf(address(stakeContract));

        vm.prank(stakerOne);
        stakeContract.alterGovernanceDelegatee(delegatee);

        uint256 contractBalanceAfter = stakingToken.balanceOf(address(stakeContract));

        //user delegatee correctly the tokens
        assertEq(contractBalanceBefore - contractBalanceAfter, stakedAmount);

        //delegates to address(0)
        vm.prank(stakerOne);
        stakeContract.alterGovernanceDelegatee(delegate0);

        vm.prank(stakerOne);
        stakeContract.alterGovernanceDelegatee(delegatee);

        contractBalanceAfter = stakingToken.balanceOf(address(stakeContract));

        //the contract's balance should remain constant since tokens are only being moved
between delegates, but currently this is not happening
        assertEq(contractBalanceBefore - contractBalanceAfter, 2 * stakedAmount);
        //rewards are decreased by 1 ether, but this value should remain constant since it
represents reward allocation
        assertEq(stakingToken.balanceOf(address(stakeContract)), 999 ether);
    }
```

Recommendations

To fix the issue, disable delegating to `address(0)`.

Medium findings

[M-01] Instant withdraw lets users self-fund residuals with their own penalty

Severity

Impact: Medium

Likelihood: Medium

Description

When a user calls `BobStaking::instantWithdraw`, the function first calls `_claimRewards(msg.sender, false)`. If `rewardTokenBalance` is insufficient, `_claimRewards` credits the shortfall to `residualRewardBalance[msg.sender]` (and may pay `rewardsToPay == 0`). Immediately after, `instantWithdraw` computes the penalty and adds it to `rewardTokenBalance`:

```
    function instantWithdraw(address _receiver) external nonReentrant {
        // ...

        _claimRewards(_stakeMsgSender(), false); #AUDIT: Here we may increase the residual
rewards

        uint256 amount = stakers[_stakeMsgSender()].amountStaked;
        if (amount == 0) revert NotEnoughBalance();

        stakingTokenBalance -= amount;

        // Calculate the penalty amount to the user
        uint256 _amountForUser = (amount * instantWithdrawalRate) / 100;
        uint256 _amountForContract = amount - _amountForUser;

        rewardTokenBalance += _amountForContract; # AUDIT: Here we increase the
rewardTokenBalance, so the before increased residual rewards, will now "self-payed", bypassing
the instant withdraw fee effectively.

        // ...
    }
```

The user can then call `claimResidualRewards` and withdraw their residual, effectively paying the instant-withdraw penalty into the pool and immediately reclaiming it as their residual payout. Net effect is that the penalty is neutralized in whole or in part (limited by the residual size), undermining the fee's purpose and shifting costs to other stakers.

In order to understand better this issue, consider this scenario : 1. Pool's `rewardTokenBalance` is near zero; Alice has accrued large unpaid rewards. 2. Alice calls `instantWithdraw(). _claimRewards(..., false)` records a big

`residualRewardBalance[Alice]`. 3. `instantWithdraw` credits the penalty to `rewardTokenBalance`. 4. Alice immediately calls `claimResidualRewards` and withdraws her residual, which is now funded by the penalty she just paid. 5. Alice's effective `penalty ≈ max(0, penalty − residual)`; if `residual ≥ penalty`, she pays no net fee.

Recommendations

It is recommended to net residuals against penalty before crediting the pool. In `instantWithdraw` after the `_claimRewards` call, do :

```
Step 1. uint256 residual = residualRewardBalance[msg.sender];
Step 2.1. If residual >= penalty: set residualRewardBalance[msg.sender] = residual - penalty;
penalty = 0;
Step 2.2. Else: penalty -= residual; residualRewardBalance[msg.sender] = 0;
Step 3. Only then: rewardTokenBalance += penalty;
```

This guarantees the user cannot recycle their penalty to pay their own residual.

[M-02] Condition setter functions are broken

Severity

Impact: Medium

Likelihood: Medium

Description

The `BobStaking::_setStakingCondition` allows the admin to alter the staking conditions via `setRewardRatios()`, `setWhitelistedDelegatees()`, and `setWhitelistedHybridNodeDelegateesViaController()`. However, if we carefully observe the `_setStakingCondition()`, it rightfully persists values for the staking conditions when passed as a parameter via the functions above:

```
    /// @dev Additional entry point for the hybrid node controller to set the whitelisted hybrid
node delegatees without going through the standard governance process
    function setWhitelistedHybridNodeDelegateesViaController(
        address[] memory _whitelistedHybridNodeDelegateesToAdd,
        address[] memory _whitelistedHybridNodeDelegateesToRemove
    ) external {
        if (!hasRole(HYBRID_NODE_CONTROLLER_ROLE, _msgSender())) revert NotAuthorized();

        StakingCondition storage condition = stakingConditions[nextConditionId - 1];
        _setStakingCondition(
            condition.baseRewardRatioNumerator,
            condition.governanceDelegationRewardRatioNumerator,
            condition.hybridNodeDelegationRewardRatioNumerator,
            new address[](0),                                                    <<@
            new address[](0),                                                    <<@
            _whitelistedHybridNodeDelegateesToAdd,                               <<@
```

```
            _whitelistedHybridNodeDelegateesToRemove                                <<@
        );
    }
```

```
    function setRewardRatios(
        uint256 _baseNumerator,
        uint256 _governanceDelegationNumerator,
        uint256 _hybridNodeDelegationNumerator
    ) external {
        if (!_canSetStakeConditions()) {
            revert NotAuthorized();
        }

        StakingCondition storage condition = stakingConditions[nextConditionId - 1];
        if (
            _baseNumerator == condition.baseRewardRatioNumerator
                && _governanceDelegationNumerator ==
condition.governanceDelegationRewardRatioNumerator
                && _hybridNodeDelegationNumerator ==
condition.hybridNodeDelegationRewardRatioNumerator
        ) {
            revert RewardRatioUnchanged();
        }
        _setStakingCondition(
            _baseNumerator,
            _governanceDelegationNumerator,
            _hybridNodeDelegationNumerator,
            new address[](0),             <<@
            new address[](0),             <<@
            new address[](0),             <<@
            new address[](0)              <<@
        );

        emit UpdatedRewardRatios(_baseNumerator, _governanceDelegationNumerator,
_hybridNodeDelegationNumerator);
    }
```

However, the `_setStakingCondition()` does not persist the values from the last condition ID, and directly assigns the parameters as provided to the latest condition ID. Hence, using `setWhitelistedHybridNodeDelegateesViaController()`, `setRewardRatios()`, and `setWhitelistedDelegatees()` does not actually add the `_whitelistedHybridNodeDelegateesToAdd` addresses to the past whitelisted enumerable set; similarly, `_whitelistedHybridNodeDelegateesToRemove` does not remove anything from the last condition, as no state was carried forward.

Recommendations

It is recommended to persist the values from the last condition ID to ensure the sanity of the current condition ID.

[M-03] DoS of staking due to unguarded receiver lock period

Severity

Impact: Medium

Likelihood: Medium

Description

The `BobStaking::stake()` and `BonusWrapper::stake()` allow users to stake in order to earn rewards and bonuses. These functions allow anyone to stake on behalf of the receiver, and if a stake contains a non-zero lock period, further stake calls would be required to use a similar lock period configuration:

```
    function stake(uint256 _amount, address receiver, uint80 lockPeriod) external nonReentrant {
        if (_amount == 0) revert ZeroTokenStake();
        // lock period must be valid
        if (!_contains(lockPeriods, lockPeriod)) {
            revert InvalidLockPeriod();
        }
        // If the user already has a lock period, the lock period supplied must be the same as
the existing lock period
        if (stakers[receiver].lockPeriod != 0 && stakers[receiver].lockPeriod != lockPeriod)
{            <<@
            revert InconsistentLockPeriod();
        }
       // . . .
    }
```

However, this logic allows for attackers to frontrun / spam potential stakers using dust amounts to set their `lockPeriod` to something unintended. For instance, if a user is willing to stake for a period of 3 months, an attacker could use a dust amount to stake for a period of 6 or 21 months. Hence, such an attack can lead to a long-term DoS of staking for a user.

Proof Of Concept

Add the following test case inside `BobStaking.t.sol`:

```
    function test_BlockStakesUsingDust() public {
        uint256 initialBalance = stakingToken.balanceOf(stakerOne);
        uint256 stakeAmount = 400 * 10 ** 18;
        uint256 dustAmount = 1; // 1 wei dust

        vm.prank(stakerOne);
        stakingToken.approve(address(stakeContract), type(uint256).max);

        // Attacker address
        address attacker = makeAddr("attacker");
        // Transfer dust to random address
        vm.prank(address(this));
        stakingToken.transfer(attacker, dustAmount);
```

```
        // approve staking contract
        vm.prank(attacker);
        stakingToken.approve(address(stakeContract), type(uint256).max);

        // Attacker frontruns / randomly stakes dust along with valid amount on behalf of
stakerOne
        vm.prank(attacker);
        stakeContract.stake(dustAmount, stakerOne, 3 * 30 days);

        // Check that stakerOne has dustAmount staked
        (uint256 amountStaked,, uint256 unlockTimestamp,,) =
stakeContract.getStakeInfo(stakerOne);
        assertEq(amountStaked, dustAmount);

        // Now stakerOne tries to stake valid amount for 6 months, but there is dust already
staked, hence fails with `InconsistentLockPeriod`
        vm.prank(stakerOne);
        vm.expectRevert(BobStaking.InconsistentLockPeriod.selector);
        stakeContract.stake(stakeAmount, stakerOne, 6 * 30 days);

    }
```

Recommendations

Option 1: It is recommended to separate each lock and implement a minimum amount of guard for staking.

Option 2: Require consent from the receiver on the first stake. E.g., enforce `receiver == _stakeMsgSender()` or accept a receiver-signed EIP-712 permit that authorizes `{receiver, lockPeriod, minAmount, deadline}`.

Low findings

[L-01] Residual claim reverts on shortage

`BobStaking::claimResidualRewards` reverts whenever `rewardTokenBalance` is lower than the user's residual allocation, while `BobStaking::_claimRewards` already supports partial payouts in the same situation. If the pool is briefly underfunded, the residual claim is blocked entirely, delaying users even though some tokens are available. Align the behavior with `_claimRewards` by transferring the available balance, zeroing it out, and leaving the remainder owed.

```
function claimResidualRewards(address receiver) external nonReentrant {
    uint256 residualBalance = residualRewardBalance[_stakeMsgSender()];
    if (rewardTokenBalance < residualBalance || residualBalance == 0) revert NotEnoughBalance();

    residualRewardBalance[_stakeMsgSender()] = 0;
    // The residual rewards come from the reward token balance
    rewardTokenBalance -= residualBalance;

    IERC20(rewardToken).safeTransfer(receiver, residualBalance);

    emit ResidualRewardsClaimed(receiver, residualBalance);
}
```

[L-02] Expired lock accepts new stake

`BobStaking::stake` lets anyone add stake to a position whose `unlockTimestamp` has already passed because the function only blocks deposits when `unbondEndTimes[receiver] != 0`. This means matured positions can be topped up and then unbonded immediately, bypassing the intended lock period for the fresh deposit. I recommend reinitialising the lock when a user adds stake to an expired position (e.g., `set unlockTimestamp = max(currentUnlock, block.timestamp + lockPeriod)`) or disallowing top-ups altogether until the user restakes from scratch.

```
// From BobStaking::stake
        if (stakers[receiver].amountStaked > 0) {
            _updateUnclaimedRewardsForStaker(receiver);
        } else {
            stakers[receiver].timeOfLastUpdate = uint80(block.timestamp);
            stakers[receiver].conditionIdOflastUpdate = nextConditionId - 1;
            stakers[receiver].lockPeriod = lockPeriod;
            stakers[receiver].unlockTimestamp = uint80(block.timestamp) + lockPeriod;
        }
```

[L-03] Missing validation allows `bonusEndTime` to be set to past timestamps

The `BonusWrapper::setBonusEndTime` lacks validation to ensure the new timestamp is in the future. An administrator could accidentally set `bonusEndTime` to a past timestamp, which would cause all next stake attempts with non-zero lock periods to revert with `BonusPeriodEnded()`:

```
function setBonusEndTime(uint256 _bonusEndTime) external onlyOwner {
    bonusEndTime = _bonusEndTime;
}
```

Consider adding validation to ensure the new timestamp is in the future (or `block.timestamp`, in order to block reward distribution).

[L-04] Missing events on key setters

`BonusWrapper::setBonusEndTime`, `BonusWrapper::setRewardOwner`, `BobStaking::setWhitelistedDelegatees`, and `BobStaking::setWhitelistedHybridNodeDelegateesViaController` update critical configuration without emitting an event. Consider emitting a dedicated event in each setter carrying the new parameters so operators and users can track configuration updates reliably.

```
function setBonusEndTime(uint256 _bonusEndTime) external onlyOwner {
    bonusEndTime = _bonusEndTime;
}

function setRewardOwner(address _rewardOwner) external onlyOwner {
    rewardOwner = _rewardOwner;
}
```

[L-05] Multistep division leads to loss of precision

The rewards calculation in `BobStaking::_calculateRewards()` uses the `TIME_UNIT` and `REWARD_RATIO_DENOMINATOR` variables, which are used to accumulate new rewards:

```
    /// @dev Calculate rewards for a staker.
    function _calculateRewards(address _staker) internal view returns (uint256 _rewards) {
           // . ..

        for (uint256 i = _stakerConditionId; i < _nextConditionId; i += 1) {
            // . . .

            uint256 rewardsSum = _rewards + ((rewardsProduct / TIME_UNIT) /
REWARD_RATIO_DENOMINATOR);      <<@

            _rewards = rewardsSum;
        }

        // Adding the boosted rate to the rewards, this is handled separately from the staking
```

```
conditions
        if (block.timestamp < boostedRateEndTime) {
            uint256 boostedRewardsProduct =
                (block.timestamp - staker.timeOfLastUpdate) * staker.amountStaked *
boostedRateNumerator;
            _rewards += ((boostedRewardsProduct / TIME_UNIT) /
REWARD_RATIO_DENOMINATOR);            <<@
        } else if (block.timestamp >= boostedRateEndTime && staker.timeOfLastUpdate <
boostedRateEndTime) {
            uint256 boostedRewardsProduct =
                (boostedRateEndTime - staker.timeOfLastUpdate) * staker.amountStaked *
boostedRateNumerator;
            _rewards += ((boostedRewardsProduct / TIME_UNIT) /
REWARD_RATIO_DENOMINATOR);             <<@
        }
```

However, as we can observe, these calculations involve multistep division, which would result in a loss of precision twice, leading to a loss of rewards for users over a longer horizon.

It is recommended to divide by `TIME_UNIT * REWARD_RATIO_DENOMINATOR` instead to ensure the precision loss happens only once.

[L-06] `withdrawRewardTokens()` allows excess withdrawal ignoring `residualRewardBalance`

When a user calls `unbond()`, if there is enough `rewardTokenBalance`, the rewards are immediately credited to the user and can be claimed once the unbonding period ends. If there isn't enough balance, the remaining rewards are stored in the `residualRewardBalance` mapping. Currently, `withdrawRewardTokens()` is implemented as:

```
function withdrawRewardTokens(uint256 _amount) external nonReentrant {
    if (!hasRole(DEFAULT_ADMIN_ROLE, _msgSender())) revert NotAuthorized();
    if (_amount > rewardTokenBalance) revert NotEnoughBalance();

    IERC20(rewardToken).safeTransfer(_msgSender(), _amount);
    rewardTokenBalance -= _amount;

    emit RewardTokensWithdrawnByAdmin(rewardTokenBalance);
}
```

The problem is that it does not account for `residualRewardBalance` of users, allowing the admin to withdraw more tokens than should be available, since some are already reserved for users.

Recommendation: Adjust `withdrawRewardTokens()` to consider the total `residualRewardBalance` of all users. One approach is to use a global `totalResidualRewardBalance` to track all users' residual balances in `_claimRewards()`.

```
function withdrawRewardTokens(uint256 _amount) external nonReentrant {
        if (!hasRole(DEFAULT_ADMIN_ROLE, _msgSender())) revert NotAuthorized();
-       if (_amount > rewardTokenBalance) revert NotEnoughBalance();
+       if (_amount + totalResidualRewardBalance > rewardTokenBalance) revert NotEnoughBalance()
```

```
        IERC20(rewardToken).safeTransfer(_msgSender(), _amount);
        rewardTokenBalance -= _amount;

        emit RewardTokensWithdrawnByAdmin(rewardTokenBalance);
    }
```

Note: `totalResidualRewardBalance` should be increased in `_claimRewards()` when rewards are assigned to residual balances and decreased in `claimResidualRewards()` when users withdraw their residual rewards.

[L-07] `TIME_UNIT` is not exactly one year

`TIME_UNIT` is used to calculate rewards based on time.

```
/// @dev Time unit for the reward ratio calculation is 1 year
uint80 public constant TIME_UNIT = 12 * 30 days;
```

The comment indicates this should represent 1 year, but it currently equals 360 days.

Recommendation: Set `TIME_UNIT` to 365 days.

[L-08] Users cannot opt out of hybrid node delegation

The `BobStaking::alterHybridNodeDelegatee` function prevents users from opting out of hybrid node delegation by enforcing whitelist validation on all new delegatee addresses, including `address(0)`:

```
function alterHybridNodeDelegatee(address newDelegatee) external nonReentrant {
    Staker storage staker = stakers[_stakeMsgSender()];
    if (staker.hybridNodeDelegatee == newDelegatee) revert DelegateeUnchanged();
    if (staker.amountStaked == 0) revert ZeroTokenStake();

    StakingCondition storage condition = stakingConditions[nextConditionId - 1];
    if (!condition.whitelistedHybridNodeDelegatees.contains(newDelegatee)) revert
DelegateeNotWhitelisted();

    _updateUnclaimedRewardsForStaker(_stakeMsgSender());

    staker.hybridNodeDelegatee = newDelegatee;

    emit HybridNodeDelegateeAltered(_stakeMsgSender(), newDelegatee);
}
```

Since `address(0)` will never be included in the whitelist, users who have set a hybrid node delegatee cannot reset it to receive only base staking rewards. Once a user opts into hybrid node delegation, they are permanently forced to delegate to one of the whitelisted nodes, even if they prefer to stop participating in this reward mechanism.

Recommendations

In `alterHybridNodeDelegatee`, consider allowing `address(0)` as a valid parameter to enable users to opt out.

[L-09] Unguarded `claimRewards()` can be leveraged to deny Instant withdrawal fees

The `claimRewards()` function allows anyone to claim rewards for the receiver due to its unguarded nature:

```
/**
 *  @notice    Claim accumulated rewards.
 *  @dev       Adds rewards to staked balance.
 */
function claimRewards(address receiver) external nonReentrant {
    _claimRewards(receiver, true);
}
```

However, this can lead to a situation where if the `rewardTokenBalance` goes down to 0 due to normal protocol function, an attacker could simply call the `claimRewards()` for every eligible receiver, which would lead to accumulation of residual rewards balance:

```
function _claimRewards(address receiver, bool shouldRevert) internal {
    _updateUnclaimedRewardsForStaker(receiver);
    if (stakers[receiver].amountStaked == 0) revert NoRewardsError();

    uint256 rewards = stakers[receiver].unclaimedRewards + _calculateRewards(receiver);

    uint256 rewardsToPay = rewards;

    if (rewards > rewardTokenBalance) {
        rewardsToPay = rewardTokenBalance;

        // add a residual reward balance for the user
        // note: penalty wont apply to this amount if instantWithdraw is called
        residualRewardBalance[receiver] += rewards - rewardTokenBalance;
        stakers[receiver].unclaimedRewards = 0;
    }
    // . . .
}
```

These residual reward balances are exempt from the instant withdrawal penalty and can be directly withdrawn via `claimResidualRewards()` when the `rewardTokenBalance` is sufficient. Hence, such an attack can allow for illicit claims where, even if the user does not intend to claim rewards, they would be led to do so forcefully, denying the protocol of the rightful instant withdrawal penalty fees.

Recommendations

It is recommended to guard the `claimRewards()` function to allow only the `msg.sender` to claim their rewards.

[L-10] In `_setStakingCondition()` sums can exceed `REWARD_RATIO_DENOMINATOR`

`_setStakingCondition()` is used to set the reward ratio numerators and assign delegates.

```
function _setStakingCondition(
        uint256 _baseNumerator,
        uint256 _governanceNumerator,
        uint256 _hybridNodeNumerator,
        address[] memory _whitelistedGovernanceDelegateesToAdd,
        address[] memory _whitelistedGovernanceDelegateesToRemove,
        address[] memory _whitelistedHybridNodeDelegateesToAdd,
        address[] memory _whitelistedHybridNodeDelegateesToRemove
    ) internal {
        uint256 conditionId = nextConditionId;
        nextConditionId += 1;

        stakingConditions[conditionId].baseRewardRatioNumerator = _baseNumerator;
        stakingConditions[conditionId].governanceDelegationRewardRatioNumerator =
_governanceNumerator;
        stakingConditions[conditionId].hybridNodeDelegationRewardRatioNumerator =
_hybridNodeNumerator;
        stakingConditions[conditionId].startTimestamp = uint80(block.timestamp);
        stakingConditions[conditionId].endTimestamp = 0;

        for (uint256 i = 0; i < _whitelistedGovernanceDelegateesToAdd.length; i++) {

stakingConditions[conditionId].whitelistedGovernanceDelegatees.add(_whitelistedGovernanceDelegateesToAdd[i]);
        }
        for (uint256 i = 0; i < _whitelistedGovernanceDelegateesToRemove.length; i++) {
            stakingConditions[conditionId].whitelistedGovernanceDelegatees.remove(
                _whitelistedGovernanceDelegateesToRemove[i]
            );
        }

        for (uint256 i = 0; i < _whitelistedHybridNodeDelegateesToAdd.length; i++) {

stakingConditions[conditionId].whitelistedHybridNodeDelegatees.add(_whitelistedHybridNodeDelegateesToAdd[i]);
        }
        for (uint256 i = 0; i < _whitelistedHybridNodeDelegateesToRemove.length; i++) {
            stakingConditions[conditionId].whitelistedHybridNodeDelegatees.remove(
                _whitelistedHybridNodeDelegateesToRemove[i]
            );
        }

        //store the time where this conditions needs to be applied
        if (conditionId > 0) {
            stakingConditions[conditionId - 1].endTimestamp = uint80(block.timestamp);
        }
    }
```

The issue is that `_baseNumerator + _governanceNumerator + _hybridNodeNumerator` is not checked against `REWARD_RATIO_DENOMINATOR`, allowing the total to exceed 100%.

This can result in rewards being calculated as more than 100% in `_calculateRewards()`.

```
uint256 rewardsProduct = (endTime - startTime) * (staker.amountStaked)
                * (
                    condition.baseRewardRatioNumerator +
governanceDelegationRewardRatioNumerator
                        + hybridNodeDelegationRewardRatioNumerator
                );

            uint256 rewardsSum = _rewards + ((rewardsProduct / TIME_UNIT) /
REWARD_RATIO_DENOMINATOR);
```

Recommendations

To fix the issue, `_setStakingCondition()` should check that `_baseNumerator + _governanceNumerator + _hybridNodeNumerator <= REWARD_RATIO_DENOMINATOR` and revert if this condition is not met.

```
function _setStakingCondition(
        uint256 _baseNumerator,
        uint256 _governanceNumerator,
        uint256 _hybridNodeNumerator,
        address[] memory _whitelistedGovernanceDelegateesToAdd,
        address[] memory _whitelistedGovernanceDelegateesToRemove,
        address[] memory _whitelistedHybridNodeDelegateesToAdd,
        address[] memory _whitelistedHybridNodeDelegateesToRemove
    ) internal {
        uint256 conditionId = nextConditionId;
        nextConditionId += 1;

+       require(_baseNumerator + _governanceNumerator + _hybridNodeNumerator =<
REWARD_RATIO_DENOMINATOR, "REWARD_RATIO_DENOMINATOR superated");

        stakingConditions[conditionId].baseRewardRatioNumerator = _baseNumerator;
        stakingConditions[conditionId].governanceDelegationRewardRatioNumerator =
_governanceNumerator;
        stakingConditions[conditionId].hybridNodeDelegationRewardRatioNumerator =
_hybridNodeNumerator;
        stakingConditions[conditionId].startTimestamp = uint80(block.timestamp);
        stakingConditions[conditionId].endTimestamp = 0;

        for (uint256 i = 0; i < _whitelistedGovernanceDelegateesToAdd.length; i++) {

stakingConditions[conditionId].whitelistedGovernanceDelegatees.add(_whitelistedGovernanceDelegat
eesToAdd[i]);
        }
        for (uint256 i = 0; i < _whitelistedGovernanceDelegateesToRemove.length; i++) {
            stakingConditions[conditionId].whitelistedGovernanceDelegatees.remove(
                _whitelistedGovernanceDelegateesToRemove[i]
            );
        }

        for (uint256 i = 0; i < _whitelistedHybridNodeDelegateesToAdd.length; i++) {

stakingConditions[conditionId].whitelistedHybridNodeDelegatees.add(_whitelistedHybridNodeDelegat
eesToAdd[i]);
        }
        for (uint256 i = 0; i < _whitelistedHybridNodeDelegateesToRemove.length; i++) {
```

```
            stakingConditions[conditionId].whitelistedHybridNodeDelegatees.remove(
                _whitelistedHybridNodeDelegateesToRemove[i]
            );
        }

        //store the time where this conditions needs to be applied
        if (conditionId > 0) {
            stakingConditions[conditionId - 1].endTimestamp = uint80(block.timestamp);
        }
    }
```

[L-11] Residual recycling lets attackers farm wrapper bonus without new capital

During the **bonus window**, `BonusWrapper.stake(...)` gives a bonus on amounts deposited through the wrapper. Because `BobStaking.claimRewards(receiver)` is callable by **anyone** and because **residual rewards** (created when `rewardTokenBalance` is low) are **paid out as liquid tokens**, an attacker can game the system:

Attack flow

1. While `rewardTokenBalance` has funds, attacker calls `claimRewards(user)` for many users → their rewards **auto-compound inside** `BobStaking` (not via `BonusWrapper`), **consuming** the reward pool and **not qualifying** for BonusWrapper's bonus.
2. Immediately after, attacker calls `claimRewards(attacker)` when the pool is now low → their rewards go to `residualRewardBalance[attacker]` (no compounding).
3. When admin later **deposits rewards**, attacker **instantly claims residuals** (now liquid ERC20).
4. Attacker **re-deposits those same tokens via** `BonusWrapper.stake` **during the bonus window**, earning the bonus **without adding new capital** (it's just already-earned rewards recycled through the wrapper).

Effects

- **Unfair bonus capture **

Recommendations

Restrict claims to self:

```
function claimRewards(address receiver) external nonReentrant {
    require(receiver == msg.sender, "Only self-claim");
    _claimRewards(receiver, true);
```

}

[L-12] Anyone can trigger others' claims causing front-running reward loss

`claimRewards(address receiver)` lets **any caller** pass any `receiver`:

```
function claimRewards(address receiver) external nonReentrant {
    _claimRewards(receiver, true);
}
```

Inside `_claimRewards`, the contract settles the **receiver's** rewards. When `rewardTokenBalance` is low, only a small portion is staked and the rest is pushed into `residualRewardBalance[receiver]` (paid later as a plain transfer, not compounded).

A malicious actor can **front-run** a user's own claim and call `claimRewards` to diminish `rewardTokenBalance` This forces the victim's rewards into `residualRewardBalance` (and zeroes their `unclaimedRewards`) right before the victim tries to auto-compound, making the victim **miss compounding yield** on the residual portion. No theft occurs, but it's a repeatable **grief** that reduces the victim's APR.

Recommendations Restrict self-claims only:** require `receiver == msg.sender`.

[L-13] Boost window mis-scaled dividing by `TIME_UNIT` not 30 days

If the intended design is "**pay a fixed total boosted amount spread over the 30-day window**" as indicated in the tests, the current implementation underpays. In `_calculateRewards`, boosted rewards are scaled by the **annual** unit `TIME_UNIT` (360 days), not by the **boost window length**:

```
// current
uint256 boostedRewardsProduct =
    (effectiveElapsedSeconds) * staker.amountStaked * boostedRateNumerator;

_rewards += ((boostedRewardsProduct / TIME_UNIT) / REWARD_RATIO_DENOMINATOR);
```

With a 30-day window (`boostedRateEndTime = now + 30 days`) and `TIME_UNIT = 12 * 30 days`, this divides by ~360 days instead of ~30 days, which the test file shows. If `boostedRateNumerator` represents a **fixed total boost over the 30-day period**, users are paid only about **1/12th (~8.3%)** of the intended boost (≈ **91.7% underpayment**).

Recommendations

Scale by the boost window length (not `TIME_UNIT`).