Berachain Pol Security Review

Auditors

Noah Marconi, Lead Security Researcher

Optimum, Lead Security Researcher

Kaden, Security Researcher

0xDjango, Associate Security Researcher

Report prepared by: Lucas Goiriz

November 22, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Berachain is an EVM-identical L1 turning liquidity into security powered by Proof Of Liquidity.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of [INSERT-PROJECT-NAME] according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, Berachain engaged with Spearbit to review the contracts-monorepo protocol. In this period of time a total of **27** issues were found.

Summary

Project Name	Berachain
Repository	contracts-monorepo
Commit	5c715a...c809
Type of Project	Staking, Yield
Audit Timeline	Sep 30th to Oct 11th
Two week fix period	Oct 11 - Oct 13

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	5	4	1
Medium Risk	5	5	0
Low Risk	6	3	3
Gas Optimizations	1	0	1
Informational	10	4	6
Total	**27**	**16**	**11**

5 Findings

5.1 High Risk

5.1.1 Call to non-existent contract allows for malicious vault creation

Severity: High Risk

Context: StakingRewards.sol#L84

Description: The `stakingToken` for new vaults is never confirmed to be a contract. Combined with the use of Solady's safeTransfer, which does not perform a contract check, a malicious actor may create a vault and obtain an unfair number of shares.

Scenario:

- `stakingToken` address is calculated but not deployed.
- A vault is created through the factory.
- The malicious actor may stake any amount of tokens, the transfer attempt via `safeTransferFrom` will silently fail but the `totalSupply` and `_accountInfo[account].balance` will be both updated.
- Deploy the `stakingToken`.
- All future stakes will be entitled to a reduced share of both rewards and deposited `stakingTokens`.

It's important to note that this allows an attacker to withdraw `stakingTokens` of others without having deposited any.

The feasibility of anticipating a token address before deployment is simple when considering tokens such as DEX pool pairs.

Exacerbating this problem, even if the malicious behavior is noticed during the whitelisting process, it would prevent a legitimate staking token from ever being used.

Recommendation: Confirm token contract exists and revert vault initialization in failure. Example below mirrors the contract check OpenZeppelin uses in their safeTransfer:

```
if (address(token).code.length == 0) {
    revert NotAContract();
}
```

For more information see: github.com/kadenzipfel/smart-contract-vulnerabilities.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by implementing the auditor's recommendation.

5.1.2 `FeeCollector.claimFees` lacks slippage protection leading to loss of `payoutAmount`

Severity: High Risk

Context: /src/pol/FeeCollector.sol#L91

Description: As documented, `FeeCollector.claimFees` draws inspiration from the Uniswap V3 Factory Owner contract. Here fees are collected in any token, remaining claimable by anyone so long as `payoutAmount` of the `payoutToken` is provided to the contract in exchange. In effect, a swap.

There is no slippage protection on this swap meaning any non-reverting but delayed transaction may lose the `payoutAmount`. A delayed or front run transaction will trigger this issue. If the fee tokens revert on zero value transfers the issues is still present so long as there is some greater than 0 amount of fee tokens remaining. As malicious actor could force the issue by returning dust to the contract after calling `claimFees`.

Recommendation: Similar to the approach adopted by Uniswap allow callers to specify minimum amounts expected per fee token. Departing from Uniswap, this will need to be in the form of an array.

Berachain: Acknowledged. We'll manage to put some proper documentation on the fact that caller should do checks on her side otherwise she will be subject to this issue.

Spearbit: The team has acknowledged the issue; however, we strongly recommend implementing the proposed solution above. If this approach is not adopted, we highly suggest including a sample contract or code snippet with the necessary slippage checks to guide bot builders.

5.1.3 `distributeFor` might be reentered leading to the loss of BGT and incentive tokens

Severity: High Risk

Context: Distributor.sol#L113-L114

Description: The code flow of `Distributor.distributeFor` is in charge of compensating validators with BGT tokens and additional vault specific tokens. During the flow of the function, it queries the `nextActionableBlock`, then it is transferring (using a potentially untrusted external call) the incentive tokens to the validator and only then increments the `getNextActionableBlock` to make sure the next transaction would not double-spend. The `distributeFor` is therefore violating the "Checks-Effects-Interactions" pattern since the external interaction is executed before the effects (incrementing the `getNextActionableBlock`). In case one of the incentive tokens implements an unsafe external call, the call flow can be hijacked to re-enter `distributeFor` repeatedly and by this drain the entire BGT and incentive tokens allocated. The reentrant call will have to provide the necessary parameters that were used originally for the call but this can be fetched by a front runner spotting the call to `distributeFor` in the mempool and writing the value to a contract that will be used through the reentrancy flow.

Recommendation: Consider either switching the order of `_distributeFor` and `_incrementBlock` so that `_incrementBlock` will be called first, or alternatively, adding a reentrancy guard to `distributeFor`. ReentrancyGuardTransient.sol might help to do it in a more gas efficient way but has partial support in EVM chains. Please refer to evmdiff.com for more information about opcodes support.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by implementing the auditor's recommendation.

5.1.4 Validators can front run calls to `addIncentive` to drain the entire incentive allocation in return for validating a single block

Severity: High Risk

Context: BerachainRewardsVault.sol#L312

Description: `addIncentive` is a function that allows anyone to allocate any amount of whitelisted incentive tokens to the `BerachainRewardsVault` contract to further incentivize validators for allocating BGT tokens to their vaults. In the current version of the code, the function allows the caller to set the incentive rate (only bounded by `MAX_INCENTIVE_RATE`) in case the amount of incentive tokens left in the contract is less than `minIncentiveRate`.

As seen in the issue "Validators can manipulate incentive rates to receive more incentives than intended", rogue validators can set the `incentiveRate` as they wish to effectively steal leftover amounts. In this issue we will describe an attack vector that uses the same vulnerability but may end up in greater damage.

Using the same parameters used in "Validators can manipulate incentive rates to receive more incentives than intended", in case `addIncentive` is called when `incentive.amountRemaining < minIncentiveRate` for a specific token (this can either happen during the first call or when the current rewards are consumed and need to be refilled). In the normal case we assume that the caller is planning to load the contract with a considerably large amount of tokens that will be consumed in accordance to his provided incentive rate.

Potential attackers can monitor the mem-pool and sandwich the call to `addIncentive` with two calls:

1. Before: call `addIncentive` with `amount=101` (an amount slightly greater than `minIncentiveRate`) and `incentiveRate=MAX_INCENTIVE_RATE`.

   Now the victim's call of `addIncentive(amount = 100,000 incentive tokens, incentiveRate = 100)` is processed, they expect it to set the incentive rate to 100, but it would not because of the "donation" just made

before, so line 329 won't be executed but the call will succeed. At this point `incentive.amountRemaining` is 100,101 incentive tokens.

```
if (amountRemaining <= minIncentiveRate && incentiveRate >= minIncentiveRate) {
        incentive.incentiveRate = incentiveRate; // line 329
}
```

2. After: call `distributeFor` and potentially siphon the entire 100,101 incentive tokens for just a single validated block since `amount` will be min(100101*1e18, 0.5 * 1e18 * 1e36 / 1e18) = 100,101 incentive tokens.

```
uint256 amount = FixedPointMathLib.mulDiv(bgtEmitted, incentive.incentiveRate, PRECISION);
uint256 amountRemaining = incentive.amountRemaining;
amount = FixedPointMathLib.min(amount, amountRemaining);
```

Recommendation: Providing a simple and hermetic solution to this issue is not an easy task. The issue stems from the fact that setting the incentive rate can be done in a permission-less way and therefore can be manipulated. While there can be a solution that will provide an efficient isolation for different callers of `addIncentive`, it will not be a simple one, but it might be inspired by the design that's used in Drips Protocol. With that being said, together with the Berachain team, we came up with a solution that is somewhat a compromise. Rather than completely solving the issue, we change the code so that attackers will only be able to temporarily DoS specific calls to `addIncentive` paying for each attack transaction slightly more than `minIncentiveRate` which will dis-incentivize them to do so. Please consider adopting this amended version of `addIncentive`, notice that it will also solve "Validators can manipulate incentive rates to receive more incentives than intended":

```
/// @inheritdoc IBerachainRewardsVault
function addIncentive(address token, uint256 amount, uint256 incentiveRate) external
↪  onlyWhitelistedToken(token) {
    if (incentiveRate > MAX_INCENTIVE_RATE) IncentiveRateTooHigh.selector.revertWith();

    Incentive storage incentive = incentives[token];
    (uint256 minIncentiveRate, uint256 incentiveRateStored, uint256 amountRemaining) =
        (incentive.minIncentiveRate, incentive.incentiveRate, incentive.amountRemaining);

    if (amount < minIncentiveRate) AmountLessThanMinIncentiveRate.selector.revertWith();

    token.safeTransferFrom(msg.sender, address(this), amount);
    incentive.amountRemaining = amountRemaining + amount;

    // if its different, then caller is trying to change the incentive rate
    // validate if change is possible if not revert to avoid front running of caller call
    if (incentiveRate != incentiveRateStored){
        // Allow to reset incentive rate if amountRemaining < minIncentiveRate
        if (amountRemaining == 0 && incentiveRate >= minIncentiveRate) {
            incentive.incentiveRate = incentiveRate;
        }

        // if reset not possible, caller trying to increase the rate, validate if possible
        else if (incentiveRate >= incentiveRateStored) {
            uint256 rateDelta;
            unchecked {
                rateDelta = incentiveRate - incentiveRateStored;
            }
            if (amount >= FixedPointMathLib.mulDiv(amountRemaining, rateDelta, incentiveRateStored)) {
                incentive.incentiveRate = incentiveRate;
            } else {
              revert();
            }
        }
        else{
          // This will break the POC as front run tx will increase `amountRemaining` just more than
↪  `minIncentiveRate`
```

```
            // also they will set the `incentiveRateStored` to max so both condition will fail and you
↪   revert
            revert();
            }
        }
        emit IncentiveAdded(token, msg.sender, amount, incentive.incentiveRate);
    }
```

Please note that the provided code is experimental and should be tested and as explained above is a practical compromise and not a holistic solution.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by making `addIncentive` permissioned. Only managers that were defined by the factory owner are allowed to call `addIncentive`. This will prevent any ability to front run this function.

5.1.5 Validators can manipulate incentive rates to receive more incentives than intended

Severity: High Risk

Context: BerachainRewardsVault.sol#L328-L330

Description: In `BerachainRewardsVault.addIncentive`, it's possible for anyone to arbitrarily set the `incentiveRate` of an incentive token given the following requirements:

- The caller must transfer in at least the `incentive.minIncentiveRate` amount of tokens.
- The `incentive.amountRemaining` must be at most the `incentive.minIncentiveRate`.
- The new `incentive.incentiveRate` must be at least the `incentive.minIncentiveRate`.

Knowing that the distribution for the block that they validated is upcoming, if the `incentive.amountRemaining` will be less than or equal to the `incentive.minIncentiveRate` but the amount of incentives the validator will receive is less than the `incentive.amountRemaining`, a validator can exploit this logic by increasing the `incentive.incentiveRate` sufficiently that they will receive the full `incentive.amountRemaining` while also being refunded the tokens that they were forced to transfer in.

Consider the following example (assuming 18 decimal tokens):

- `incentive.minIncentiveRate` = `100e18`
- `incentive.incentiveRate` = `100e18`
- `incentive.amountRemaining` = `99e18`
- `bgtEmitted` = `0.5e18`

Normally, in this case the validator can expect to receive `bgtEmitted * incentive.incentiveRate == 0.5e18 * 100e18 == 50e18` tokens. However, if the validator were to instead call `addIncentive` with an `incentiveRate` of 398e18, transferring in 100e18 tokens, they would instead receive a total of `bgtEmitted * incentive.incentiveRate == 0.5e18 * 398e18 == 199e18` tokens, effectively receiving a full refund for the tokens they had to transfer in to increase the `incentive.incentiveRate` and also receiving the full `incentive.amountRemaining` of 99 tokens, receiving 44 tokens more than expected.

Recommendation: To prevent this attack, it's necessary to do one of the following:

- Only allow arbitrary changes to the `incentive.incentiveRate` if the `incentive.amountRemaining` is 0:

```
-i f (amountRemaining <= minIncentiveRate && incentiveRate >= minIncentiveRate) {
+ if (amountRemaining == 0 && incentiveRate >= minIncentiveRate) {
        incentive.incentiveRate = incentiveRate;
  }
```

- Or only allow for the `incentive.incentiveRate` to be decreased in this case:

```
  -i f (amountRemaining <= minIncentiveRate && incentiveRate >= minIncentiveRate) {
  + if (amountRemaining <= minIncentiveRate && incentiveRate >= minIncentiveRate && incentiveRate
↪  < incentiveRateStored) {
        incentive.incentiveRate = incentiveRate;
    }
```

- Or enforce that the `amount` provided is at least the new `incentiveRate`:

```
  - if (amount < minIncentiveRate) AmountLessThanMinIncentiveRate.selector.revertWith();
  + if (amount < incentiveRate) AmountLessThanMinIncentiveRate.selector.revertWith();
```

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by making `addIncentive` permissioned. Only managers that were defined by the factory owner are allowed to call `addIncentive`. This will prevent any ability to front run this function.

5.2 Medium Risk

5.2.1 Activation of queued cutting board can be manipulated leading to redirection of BGT

Severity: Medium Risk

Context: Berachef.sol#158

Description: A validator operator can queue a new cutting board at any time. Once the `cuttingBoardBlockDelay` has passed, the queued cutting board is ready for activation. The activation of a cutting board occurs via `distributor.distributeFor()` which calls `beraChef.activateReadyQueuedCuttingBoard(pubkey, blockNumber);`.

The validator is incentivized to emit the BGT reward to reward vaults that will provide the best financial incentives while also incentivizing BGT holders to stake their BGT to the validator to boost emissions.

However, a malicious validator can game this system by publicly queueing a cutting board to incentivize certain reward vaults. This will attract the BGT stakes of interested stakeholders. When the validator is chosen to validate a block, the validator can queue a new cutting board which will invalidate the previously queued cutting board that would be activated and used for emissions when calling `distributor.distributeFor()`.

Example:

- Validator currently has weights set to three reward vaults A, B, and C.
- Validator queues to instead cut rewards to D, E, and F. The `startBlock` is set at 1000.
- Protocols benefitting from D, E, and F shift support to Validator based on proposed cutting board.
- Blocks pass, now on block 1100.
- Validator gets selected for block.
- Someone calls distributeFor().
- Validator frontruns and calls queueNewCuttingBoard(). It doesn't matter which weights they select. Simply, it will set the `queuedCuttingBoard[valPubKey.startBlock]` in the future.
- `isQueuedCuttingBoardReady()` will return false and short-circuit `activateReadyQueuedCuttingBoard()`.
- The rewards will still be cut to A, B, and C.

Recommendation: Do not allow validators to queue a new cutting board if the currently queued cutting board is ready for activation.

Berachain: Fixed in commit 62c6d466.

Spearbit: Fixed by removing the ability to queue another cutting board if an operator already has a queued cutting board.

5.2.2 Updating `FeeCollector` `payoutToken` or `payoutAmount` can result in loss to fee collector or protocol

Severity: Medium Risk

Context: FeeCollector.sol#L73-L84

Description: In `FeeCollector.claimFees`, there is a keeper mechanism wherein when the value of fee tokens exceeds the `payoutAmount` + associated gas costs, a keeper will call the function to profit the difference. In the case that the `payoutToken` or `payoutAmount` are changed in `setPayoutToken`/`setPayoutAmount`, the value of the `payoutAmount` will likely increase or decrease.

If the value of the `payoutAmount` decreases from one of these changes, it could suddenly make `claimFees` highly profitable, leading to a keeper calling the function, thereby causing the system to overpay for these fees to be claimed. Similarly, although less likely, if the value of the `payoutAmount` increases from one of these changes while a keeper has already submitted a transaction to call `claimFees`, it can result in a loss for the keeper.

Recommendation: To prevent this, it's important that we don't update the `payoutAmount` or `payoutToken` if there are remaining fee tokens. A good solution to this may be to set pending values for the `payoutAmount`/`payoutToken`, which only get updated at the end of `claimFees`.

Note that it's technically still possible for this problem to occur with an extreme change to the `payoutAmount` if other fee tokens are leftover, but it's highly unlikely since in an efficient market, all fees will be claimed as soon as it's profitable to do so.

Berachain: Fixed in PR 441.

Spearbit: Fixed by modifying changes to `payoutAmount` as recommended and by removing `setPayoutToken`, effectively making the `payoutToken` immutable.

5.2.3 An update to `StakingRewards.rewardToken` allows the `owner` to drain all `rewardTokens`

Severity: Medium Risk

Context: /src/base/StakingRewards.sol#L38-L41, /src/pol/BGTStaker.sol#L81-L85

Description: `StakingRewards` contract leaves a number of variables mutable, namely the `rewardToken` which has an exposed setter in `BGTStaker.sol`.

After updating `rewardToken`, all unclaimed rewards become transferable to owner by calling `BGTStaker.rewardToken`.

Recommendation: Do not allow updating of reward tokens.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by implementing the auditor's recommendation.

5.2.4 Calls to `StakingRewards._setRewardsDuration` might always revert since `periodFinish` is constantly increasing

Severity: Medium Risk

Context: StakingRewards.sol#L238

Description: `_setRewardsDuration` is supposed to allow the owner to set the value of `rewardsDuration` which affects the value of `periodFinish` which in turn affects the calculation of `rewardPerToken`.

```
function _setRewardsDuration(uint256 _rewardsDuration) internal virtual {
    // TODO: allow setting the rewards duration before the period finishes.
    if (_rewardsDuration == 0) RewardsDurationIsZero.selector.revertWith();
    if (block.timestamp <= periodFinish) RewardCycleNotEnded.selector.revertWith();
    rewardsDuration = _rewardsDuration;
    emit RewardsDurationUpdated(_rewardsDuration);
}
```

As we can see, the function will revert as long as `block.timestamp <= periodFinish`, i.e. during an active period. While it makes sense to add this condition, in practice it will be hard to find a time slot in which the function won't revert for this condition since `_setRewardRate` is expected to be called regularly (every few seconds), increasing `periodFinish` by `rewardsDuration` (which is set to 7 days by default).

Recommendation: Consider removing the line of:

```
if (block.timestamp <= periodFinish) RewardCycleNotEnded.selector.revertWith();
```

alongside with the TODO comment at the beginning of the function to allow the owner to set the rewards duration even before the period finishes.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by implementing the auditor's recommendation.

5.2.5 `addIncentive` front runners can silently block any attempt to increase the incentive rate

Severity: Medium Risk

Context: BerachainRewardsVault.sol#L340-L342

Description: Front runners that spot calls to `addIncentive` that are meant to add an amount of incentive tokens to the vault contract while setting the `incentiveRate` as well can block the expected increase by calling `addIncentive` right before donating a dust amount of tokens yet large enough to cause line 341 to not be executed.

```
if (amount >= FixedPointMathLib.mulDiv(amountRemaining, rateDelta, incentiveRateStored)) {
    incentive.incentiveRate = incentiveRate; // line 341
}
```

Recommendation: This issue is not easily solvable. We recommend on keeping this code but instead of letting it silently skip line 341, consider reverting instead. This solution will means practically that callers should take a "slippage" that should be added on top of the provided `amount` to increase the chance that their transaction will be able to set the value of `incentive.incentiveRate` without reverting. Please refer to the code snippet provided in the issue "Validators can front run calls to `addIncentive` to drain the entire incentive allocation in return for validating a single block".

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by making `addIncentive` permissioned. Only managers that were defined by the factory owner are allowed to call `addIncentive`. This will prevent any ability to front run this function.

5.3 Low Risk

5.3.1 `BeaconDeposit.deposit`: frontrunners can cause the original caller's transaction to revert

Severity: Low Risk

Context: BeaconDeposit.sol#L78

Description: *(This issue was found as part of an additional review that focused on newly introduced PRs (453, 461, 463), indirectly related to the original review).*

Frontrunners may detect `deposit` calls and attempt to front-run them by initiating their own calls with the same pubkey, either with the original operator or an alternative one. This tactic can cause the original transaction to revert, temporarily blocking the victim's deposit attempt, although the original caller's funds remain secure. If repeated, this can result in a continuous but not permanent denial of service for the victim, requiring them to generate a new pubkey and resubmit the transaction. Notably, attackers must stake at least `MIN_DEPOSIT_AMOUNT_IN_GWEI` in BERA tokens to launch, which is intended to discourage such front-running attempts.

Recommendation: At the moment, there is no simple solution to mitigate it. As discussed with the team, once BLS precompiled contracts are more mature, it will be solved by using a BLS signature. In the meantime, it is

important to solve this issue in the frontend level, which means to catch this case and communicate it well to the user.

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.3.2 Updating `activateBoostDelay` retroactively affects all queued boosts

Severity: Low Risk

Context: BGT.sol#L208

Description: Users queue their boosts using `queueBoost()`. Once enough time has passed, their boost can be activated by calling `activateBoost()`. The amount of time that they must wait is stored in variable `activateBoostDelay`.

This variable can be updated by the admin at any time, meaning that previously queued boosts will be affected retroactively by the new value. This also affects `dropBoostDelay`.

Recommendation: Consider storing the value of `activateBoostDelay` and `dropBoostDelay` at the time of queuing the boost and later retrieve it for use in the `_checkEnoughTimePassed()` check. This will ensure that users are not forced to wait longer than they originally agreed to.

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.3.3 Batched calls to `activateBoost()` can be DOSed

Severity: Low Risk

Context: BGT.sol#28

Description: The BGT contract inherits `Multicallable.sol` so that calls to `activateBoost()` can be batched together in a single transaction. However, this pattern is susceptible to griefing as a failure to a single boost activation will cause the entire transaction to revert. If the reversion is toward the end of the batched calls, this may result in significant gas spend.

The issue lies in the fact that `activateBoost()` reverts if the queued boost cannot be activated at the current timestamp. Therefore, a malicious user can queue their boost, wait for batched calls to activate their boost, and re-queue a new boost to reset their `blockNumberLast`.

```
function activateBoost(address user, bytes calldata pubkey) external {
    QueuedBoost storage qb = boostedQueue[user][pubkey];
    (uint32 blockNumberLast, uint128 amount) = (qb.blockNumberLast, qb.balance);
    // `amount` zero will revert as it will fail with stake amount being zero at `stake` call.
    _checkEnoughTimePassed(blockNumberLast, activateBoostDelay);
```

```
function _checkEnoughTimePassed(uint32 blockNumberLast, uint32 blockBufferDelay) private view {
    unchecked {
        uint32 delta = uint32(block.number) - blockNumberLast;
        if (delta <= blockBufferDelay) NotEnoughTime.selector.revertWith();
    }
}
```

```
function queueBoost(bytes calldata pubkey, uint128 amount) external checkUnboostedBalance(msg.sender,
↪  amount) {
    userBoosts[msg.sender].queuedBoost += amount;
    unchecked {
        QueuedBoost storage qb = boostedQueue[msg.sender][pubkey];
        // `userBoosts[msg.sender].queuedBoost` >= `qb.balance`
        // if the former doesn't overflow, the latter won't
        uint128 balance = qb.balance + amount;
        (qb.balance, qb.blockNumberLast) = (balance, uint32(block.number));
    }
    emit QueueBoost(msg.sender, pubkey, amount);
}
```

Recommendation: Do not revert, simply return early.

Berachain: Fixed in commit 20517616.

Spearbit: Fixed by returning `true` or `false` in the time difference check instead of reverting.

5.3.4 `FeesClaimed` event emission can be spoofed to show misleading amount of fees claimed

Severity: Low Risk

Context: FeeCollector.sol#L101

Description: When profitable, anyone is able to call `FeeCollector.claimFees()` which will transfer `payoutAmount` of the `payoutToken` to the reward receiver. In return, the caller receives the contract's balance of the specified `_feeTokens`. At the end of execution, an event is emitted `emit FeesClaimed(msg.sender, _recipient, feeToken, feeTokenAmountToTransfer);`.

This event is easily spoofed by sending arbitrary amounts of `feeToken` to the contract prior to calling `claimFees()`. This will cause the event to emit that the caller received a large reward for the usual amount of payout token. Events are often used off-chain to index occurrences on the blockchain and are relied upon to paint an accurate depiction.

Recommendation: None at this time.

Berachain: Acknowledged. Could be a minor problem indeed, but fix (if found) may not be worth it.

Spearbit: Acknowledged.

5.3.5 Missing `__gap` variable in base contracts used for upgradeable derived contracts

Severity: Low Risk

Context: StakingRewards.sol#L17, RootHelper.sol#L9

Description: Both `RootHelper` and `StakingRewards` are abstract contracts that use the regular linear storage pattern of solidity. These are also used in the context of upgradeable derived contracts, i.e. the underlying storage will be stored in the proxy and not in the implementation itself upon deployment. The issue here is that in case of an upgrade that requires addition of storage variables to these base contracts, any storage variable that will be added in the end of their storage layout will override existing storage variables in the derived contracts.

Recommendation: Consider either adding a `uint256[50] __gap` variable to both these contracts in the end of their storage layout or alternatively change the storage allocation to be based on EIP-1967 (instead of linear storage) like in the inherited contract `OwnableUpgradeable.sol`. In addition, it is highly recommended to use a plugin like openzeppelin-upgrades that ensures future upgrades of contracts are safe, for foundry support use integrating-with-hardhat section of the Foundry book.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by adding a `__gap` variable in the end of the storage layout.

5.3.6 `BeraChef`: `maxNumWeightsPerCuttingBoard` might not represent the actual maximum number of weights

Severity: Low Risk

Context: BeraChef.sol#L108

Description: `setMaxNumWeightsPerCuttingBoard` can be used by the owner of `BeraChef` to set (increase of decrease) `maxNumWeightsPerCuttingBoard`. This value is used as the upper bound of new cutting boards and a new default cutting board, cutting boards with weights arrays that are longer than `maxNumWeightsPerCuttingBoard` will not be permitted. The issue however is that in case `setMaxNumWeightsPerCuttingBoard` is used for a decrease, the new value of `maxNumWeightsPerCuttingBoard` might be less than the actual length of the stored weight arrays for the default and current cutting boards.

Recommendation: Consider changing `setMaxNumWeightsPerCuttingBoard` so that in case of a decrease of `maxNumWeightsPerCuttingBoard`, the caller will have to provide a new cutting board to set the default cutting board in case it has more receivers than the new maximum value. Either way, `getActiveCuttingBoard` should be changed to return the `defaultCuttingBoard` in case the current value of `maxNumWeightsPerCuttingBoard` is less than the length of the active cutting board of that operator. This condition alongside with the scenario of reverting to the default cutting board in case of a receiver removed from `isFriendOfTheChef` should be documented properly and you might even consider emitting an event in case `getActiveCuttingBoard` is returning the `defaultCuttingBoard` so that operators will be notified when their emissions are directed to the default list instead.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by reverting `setMaxNumWeightsPerCuttingBoard` in case `_maxNumWeightsPerCuttingBoard < defaultCuttingBoard.weights.length`.

5.4 Gas Optimization

5.4.1 Gas optimizations

Severity: Gas Optimization

Description: See each case below

1. BerachainRewardsVault.sol#L182: `removeIncentiveToken` is used to remove incentive tokens previously whitelisted. The function is callable by the factory vault manager only but is not transferring any leftover amounts that reside in the contract. It is important to mention that `recoverERC20` can be used for that purpose but it will require a different call that can be saved. Consider transferring the leftover balance of the removed incentive token to an address controlled by the bera team by calling `token.trySafeTransfer`.
2. BerachainRewardsVault.sol#L409: `whitelistedTokens` should be an EnumerableSet.sol so that calling `_deleteWhitelistedTokenFromList` will require time complexity of $\mathcal{O}(1)$.
3. BlockRewardController.sol#L156:

   ```
   uint256 tmp_0 = uint256(FixedPointMathLib.powWad(boost, rewardConvexity));
   if (tmp_0 == one) {...}
   ```

   the calculation of `FixedPointMathLib.powWad(boost, rewardConvexity)` can be saved and `tmp_0 == one` can be changed to `rewardConvexity == 0`.
4. BeraChef.sol#L239: Consider returning the value of `startBlock` alongside with the bool to avoid an extra hot read as part of `activateReadyQueuedCuttingBoard`.
5. BeraChef.sol#L158: The function can be optimized to use a single iteration (instead of two). It is important to mention that the tradeoff here will be to compromise on code reuse.
6. StakingRewards.sol#L224: `_updateReward` mirrors the Synthetix implementation and inherits its duplicate functions calls and storage loads. To some extent the optimizer will assist in caching the duplication but there would be gas savings in a refactor to read and call necessary storage/functions 1 time only. <!-(@bronicle pasted a diagram in the comments but likely not a good fit for the issue)-->

7. BerachainRewardsVaultFactory.sol#L49-L54, BlockRewardController.sol#L57-L62 and others sharing similar initialize arguments. All of the initialization arguments are known at deploy time and with the exception of `_governance` do not change over time. `_bgt`, `_distributor`, and `_beaconDepositContract` would be better suited as immutable. The `address beacon` itself can also be cached as an immutable within the constructor.
8. BerachainRewardsVaultFactory.sol#L27-L36, FeeCollector.sol#L35, BeraChef.sol#L30-L34, BlockRewardController.sol#L29-L35, Distributor.sol#L38-L45: there are storage variables used in implementation contracts which cannot be modified, effectively making them immutable. We can make these variables immutable by initializing them in the implementation contract constructor as long as we know them at the time of deployment. This will work even via a proxy contract because the immutables from the implementation contract will be referenced.
9. BerachainRewardsVaultFactory.sol#L85-L89: instead of using the hash of the `stakingToken` in `BerachainRewardsVaultFactory.createRewardsVault`, we can use the `stakingToken` address directly (after casting to `bytes32`). This is just as secure since we can't reuse the same `stakingToken` regardless. Note that `predictRewardsVaultAddress` must also be modified to match this change.
10. BerachainRewardsVault.sol#L335: in `BerachainRewardsVault.addIncentive`, we have an `else if` block where we update the `incentive.incentiveRate` only if the provided `incentiveRate` is greater than or equal to the existing `incentiveRateStored`. We should instead only update this value if the `incentiveRate` is strictly greater than the `incentiveRateStored` because it's redundant to set the `incentive.incentiveRate` as the same value which it already is.
11. BerachainRewardsVault.sol#L50-L54: in the `Incentive` struct, we include both `minIncentiveRate` and `incentiveRate` as type uint256. Knowing that these values must be less than the `MAX_INCENTIVE_RATE` of `1e36`, we can pack these fields into type uint128's (max value of $2^{128} - 1 = 3.4e38$). This allows the two fields, which would have otherwise taken two separate storage slots, to both fit inside a single storage slot, saving one `SLOAD`/`SSTORE` each time they're read from or written to.

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 Unverified `BeaconDeposit.deposit` parameters can lead to lost deposits

Severity: Informational

Context: BeaconDeposit.sol#L78-L122

Description: `BeaconDeposit` is based off of the original Ethereum beacon chain `DepositContract`. In this original `DepositContract`, a `deposit_data_root` parameter is included in the `deposit` function. This parameter is used like a checksum to validate that the other parameters are correctly provided. This is valuable because the `deposit` function does not actually validate the `signature`, and in the case that the `signature` is incorrectly provided for a new validator, the `deposit` call will not revert but the actual beacon chain deposit will fail, causing the funds to be permanently lost.

`BeaconDeposit` does not contain this `deposit_data_root` parameter and verification logic, making users susceptible to this loss of funds in the case that their `signature` is incorrectly provided.

Recommendation: Implement the `deposit_data_root` parameter and verification logic as is used in the original `DepositContract`.

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.5.2 `Utils`: "memory-safe" assembly blocks may not conform with Solidity memory safety requirements

Severity: Informational

Context: Utils.sol#L44-L60

Description: (*This issue was found as part of an additional review that focused on newly introduced PRs (453, 461, 463), indirectly related to the original review.*)

As described in soliditylang#memory-safety, assembly blocks that marked as "memory-safe" are not allowed to write values that don't represent the next free space in memory. A quick recap for the beginning of the memory layout:

```
0x00 - 0x3f (64 bytes): scratch space for hashing methods
0x40 - 0x5f (32 bytes): a.k.a. free memory pointer
0x60 - 0x7f (32 bytes): zero slot
```

free memory pointer points to `0x80` initially.

Both versions of `revertWith` with the 3 parameters do not comply with the requirements mentioned in the solidity-lang document stating that:

> Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates.

It is important to mention that the severity of this specific issue is informational since the cases mentioned are not exploitable due to the revert. However, we filed this issue to raise awareness for memory potentially unsafe manipulation operations that may be hard to detect and may lead to an unintended behavior.

Recommendation: Consider using the free memory pointer for storing new values in memory instead of overwriting its value.

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.5.3 Pause `donate` along with `claimFees`

Severity: Informational

Context: FeeCollector.sol#L110

Description: `FeeCollector.claimFees` has the `whenNotPaused` modifier, causing calls to revert when the contract is paused. `FeeCollector.donate` does not have this modifier even though the too functions share some significant logic, notably the ability to call `BGTStaker.notifyRewardAmount`. While there's no clear reason to believe that `donate` is unsafe, it may be worth pausing `donate` along with `claimFees` since `notifyRewardAmount` is a powerful function.

Recommendation: Consider adding the `whenNotPaused` modifier to `FeeCollector.donate`:

```
- function donate(uint256 amount) external {
+ function donate(uint256 amount) external whenNotPaused {
      // donate amount should be at least payoutAmount to notify the reward receiver.
      if (amount < payoutAmount) DonateAmountLessThanPayoutAmount.selector.revertWith();

      // Directly send the fees to the reward receiver.
      payoutToken.safeTransferFrom(msg.sender, rewardReceiver, amount);
      BGTStaker(rewardReceiver).notifyRewardAmount(amount);

      emit PayoutDonated(msg.sender, amount);
  }
```

Berachain: Fixed in PR 431.

Spearbit: Fixed as recommended.

5.5.4 Validate all token addresses have code or make use of OZ's `safeTransfer`

Severity: Informational

Context: StakingRewards.sol#L19, BGTStaker.sol#L21, FeeCollector.sol#L22, BerachainRewardsVault.sol#L31, *(out of scope files not noted but some do make use of the same pattern)*.

Description/Recommendation: When using Solady `SafeTransferLib` and not validating a token has code deployed to its address at the time of transfer (contrasting with the OZ `safeTransfer` behavior), it is important to validate elsewhere that tokens have code deployed at their address.

While this issue is information in nature, when combined with other aspects of the code base, vulnerabilities can surface.

Related to the issue "Call to non-existent contract allows for malicious vault creation".

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.5.5 Benefit in adding deployment configuration checks

Severity: Informational

Context: BGTStaker.sol#L39-L43, StakingRewards.sol#L76-L79, BerachainRewardsVaultFactory.sol#L49-L55, BeraChef.sol#L59-L65,BlockRewardController.sol#L57-L62, Distributor.sol#L55-L61, BGTStaker.sol#L39-L44

Description: `FeeCollector` has a constructor with zero address checks on constructor args but most of the remaining files in the system do not.

While there are deploy scripts to reduce the likelihood of error, adding these checks would add safety with minimal overhead.

Recommendation: Consider zero address checks on constructor or initialization functions. Could go as far as adding a sentinel-like value to ensure the correct contract is passed in (i.e. `governance.sentinel() returns (bytes32)`).

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.5.6 Inconsistent use of modifier vs internal function

Severity: Informational

Context: /src/base/StakingRewards.sol#L130, /src/base/StakingRewards.sol#L105

Description: Departures from the Synthetix implementation for the `StakingRewards._stake` and `StakingRewards._withdraw` functions necessitate calling `_updateReward` as an internal function. While `_notifyRewardAmount` and `_getReward` use the old pattern of applying a modifier.

Recommendation: Given there are already changes taking place, and state changes within modifiers are not generally recommended, consider using the internal function and eliminating the modifier.

Berachain: Acknowledged.

Spearbit: Acknowledged.

5.5.7 `BerachainRewardsVaultFactory.createRewardsVault` can be front-ran only leading the original call to revert but the original state changes will persist

Severity: Informational

Context: BerachainRewardsVaultFactory.sol#L82

Description: Callers of `createRewardsVault` might see their transaction is failing on the `VaultAlreadyExists` error for a given `stakingToken`. The cause of it might be and intentional or unintentional front-running of their transaction. It is important to mention that the `vault` corresponding to the provided `stakingToken` will be deployed as expected by the front-runner and that besides of causing a potential confusing for the original caller, there is no further damage that can be made.

Recommendation: Make sure that this edge case is well documented in `IBerachainRewardsVaultFactory` so that the caller (especially in case of a smart contract caller) will be aware of this limitation. In addition, consider changing line 82 to:

```
address cachedAddress = getVault[stakingToken];
if (cachedAddress != address(0)) return cachedAddress;
```

This will make sure that the transaction does not revert so it won't cause a denial of service for the caller. Please note that this change will impact EOA callers that will need to query the value of `getVault[stakingToken]` in case their transaction does not include the `VaultCreated` event.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by implementing the auditor's recommendation.

5.5.8 `BeraChef._validateWeights`: `weights` array may include duplicated receivers

Severity: Informational

Context: BeraChef.sol#L265-L275

Description: `_validateWeights` is called upon adding a new cutting board by an operator or when the owner of the contract is setting the default cutting board. it makes sure that the sum of all weights is 100% but does not check for duplicates. Although it does not create any direct exploitable issue, we think it might be worth adding this extra check.

Recommendation: Consider sorting the weights array by the field of `weight.receiver` off-chain and then verify on-chain that the array is indeed sorted in strictly ascending order during the loop in `_validateWeights`.

Berachain: Acknowledged. Won't handle this since would add ux changes on our side.

Separbit: Acknowledged.

5.5.9 `BeraChef`: Operators should monitor `FriendsOfTheChefUpdated` events

Severity: Informational

Context: BeraChef.sol#L212

Description: Operators can use `queueNewCuttingBoard` to choose their list of vaults they want to incentivize with BGT tokens. Their set of vaults is limited to pre-approved addresses stored in `isFriendOfTheChef` by the contract owner. However, in case an address is removed from `isFriendOfTheChef` by the owner while still being part of some active cutting board for operator(s) it will cause the next call to `getActiveCuttingBoard` made by the `Distributor` contract to use the `defaultCuttingBoard` which is not necessarily optimal for operators.

Recommendation: Consider documenting the fact that operators should implement a script that monitors `FriendsOfTheChefUpdated` events to potentially change their active cutting boards in accordance.

Berachain: Acknowledged. Going to add proper documentation for validator operators.

Spearbit: Acknowledged.

5.5.10 Code quality comments

Severity: Informational

Context: See each case below

Description:

1. BerachainRewardsVault.sol#L78: Consider replacing `Incentive incentives` with just `Incentive` since the current name is confusing, and may be interpreted as that the value of the mapping is an array while it is not.
2. BerachainRewardsVault.sol#L277: inline comment should be: "underflow is impossible because `info.delegateTotalStaked` >= `stakedByDelegate` >= `amount`".
3. BerachainRewardsVault.sol#L315: Consider renaming `amountRemaining` to `amountRemainingBefore` to highlight the difference between this variable and `incentive.amountRemaining` that's updated later.
4. BlockRewardController.sol#L112: Consider replacing the usage of `ether` with `FixedPointMathLib.WAD`.
5. StakingRewards.sol#L220: Consider removing the comment of "// TODO: remove undistributedRewards".
6. StakingRewards.sol#L55, StakingRewards.sol#L236, BerachainRewardsVault.sol#L372: Address or remove remaining TODOs (Out of scope directories contain additional TODOs not noted here).
7. StakingRewards.sol#L271: returns a value scaled by `PRECISION`. Since it is a public view function you may want to at least add an inline comment about it to make it easier for external users to integrate.
8. Distributor.sol#L104: `blockNumber` is equivalent to `nextActionableBlock` and therefore is not needed as a function parameter.
9. BGT.sol#L126: Avoid variable shadowing of `owner` as both storage variable and functional argument.
10. BeraChef.sol#117: `cuttingBoardBlockDelay` can be set in the initializer to avoid have to set this value explicitly via `setCuttingBoardBlockDelay()`.
11. BerachainRewardsVault.sol#L298-L303: `recipient` only pertains to _getReward(). Consider adding a comment to avoid confusion that recipient will receive stakeToken from _withdraw().
12. StakingRewards.sol#L248: Similar calculations in the contract multiply rate * duration. Consider swapping `remainingTime` and `rewardRate` for consistency.

Berachain: Fixed in commit 534a0eba.

Spearbit: Fixed by implementing the auditor's recommendation.