SHERLOCK

Security Review For Roots

Collaborative Audit Prepared For:	**Roots**
Lead Security Expert(s):	**TessKimy**
Date Audited:	**August 26 - August 29, 2025**
Final Commit:	**2323ac0**

Introduction

This security review focuses on the implementation, utilization and redemption of rBGT which allows roots to sustainably offer high yield on assets and ensure capital efficiency with zero interest rate loans. The goal is to ensure flexibility for user via rBGT and capital efficiency via MEAD, while roots gives us revenue on the user basis and utilizes proof of liquidity via their incentives system.

Scope

Repository: roots-fi/roots-core

Audited Commit: fa0a88a9893a6321168c21e7c0022cfadc1398f9

Final Commit: 2323ac06e677adc9ce6961308ff646f152a90d1e

Files:

- src/core/BGTHandler.sol
- src/core/RootsBGT.sol
- src/interfaces/IBGTIncentiveDistributor.sol

Final Commit Hash

2323ac06e677adc9ce6961308ff646f152a90d1e

Findings

Each issue has an assigned severity:

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

Issues Found

High	Medium	Low/Info
4	4	2

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Almost any storage variable can be corrupted due to incorrect assumption

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/4

Summary

In current boosting system, all the validators are tracked with internal storage system. We update these variables after following BGT function calls:

`queueBoost` `cancelBoost` `activateBoost` `queueDropBoost` `dropBoost`

We assume that only owner can execute these operations but that's not correct. External malicious actors can execute `activateBoost` and `dropBoost` functions and they can corrupt entire internal storage system for validators.

Vulnerability Detail

We have `onlyOwner` modifier and only owner can execute these function in the BGT Handler contract. However, malicious actor can directly call `BGT` contract with `activateBoost` and `dropBoost` functions because it's permissionless on BGT side.

Following function is directly fetched from BGT implementation:

https://vscode.blockscan.com/80094/0x656b95E550C07a9ffe548bd4085c72418Ceb1dba

```
function activateBoost(address user, bytes calldata pubkey) external returns (bool)
↪  {
    QueuedBoost storage qb = boostedQueue[user][pubkey];
    (uint32 blockNumberLast, uint128 amount) = (qb.blockNumberLast, qb.balance);
    // `amount` must be greater than zero to avoid reverting as
    // `stake` will fail with zero amount.
    if (amount == 0 || !_checkEnoughTimePassed(blockNumberLast,
    ↪  activateBoostDelay)) return false;

    totalBoosts += amount;
    unchecked {
        // `totalBoosts` >= `boostees[validator]` >= `boosted[user][validator]`
        boostees[pubkey] += amount;
        boosted[user][pubkey] += amount;
        UserBoost storage userBoost = userBoosts[user];
        (uint128 boost, uint128 _queuedBoost) = (userBoost.boost,
        ↪  userBoost.queuedBoost);
        // `totalBoosts` >= `userBoosts[user].boost`
        // `userBoosts[user].queuedBoost` >= `boostedQueue[user][validator].balance`
        (userBoost.boost, userBoost.queuedBoost) = (boost + amount, _queuedBoost -
        ↪  amount);
    }
    delete boostedQueue[user][pubkey];

    IBGTStaker(staker).stake(user, amount);

    emit ActivateBoost(msg.sender, user, pubkey, amount);
    return true;
}
```

As you can see only pre-condition to execute this function is enough block should be passed, BGT token doesn't check who is the caller in `activateBoost` function. (Same as dropBoost).

Impact

This is a critical bug because after several direct calls through BGT contract, our entire validator storage system will be corrupted. It will cause following impact:

1. Stuck funds in BGT staker
2. Unintended function reverts due to underflow
3. Entire system corruption

Code Snippet

https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/blob/db5f1ccebd639a058e65665bce9d55c2b8df6d48/roots-core/src/core/BGTHandler.sol#L337-L341

https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/blob/db5f1ccebd639a058e65665bce9d55c2b8df6d48/roots-core/src/core/BGTHandler.sol#L388-L391

Tool Used

Manual Review

Recommendation

I have two possible solution for this case:

1. Using BGT contract's up-to-date storage system instead of internal tracking
2. Implementing a `syncValidator` function and calling it before each function (for every function)

I actually suggest first option because it's safest and cheapest way to handle it but unfortunately, it needs big changes on BGT Handler contract.

Second solution is a not a good one because depends on validator count it can make the system very expensive in terms of gas spending, moreover it may cause DoS if it reachs block gas limit.

An example for solution 1

BGT contract already tracks the keys (validators) as we did in our contract.

```
// BGT implementation
    mapping(address account => mapping(bytes pubkey => QueuedBoost)) public
     ↪  boostedQueue;

    /// @notice The mapping of queued drop boosts on a validator by an account
    mapping(address account => mapping(bytes pubkey => QueuedDropBoost)) public
     ↪  dropBoostQueue;

    /// @notice The mapping of balances used to boost validator rewards by an
     ↪  account
    mapping(address account => mapping(bytes pubkey => uint128)) public boosted;

    /// @notice The mapping of boost balances used by an account
    mapping(address account => UserBoost) internal userBoosts;

    /// @notice The mapping of boost balances for a validator
    mapping(bytes pubkey => uint128) public boostees;
```

We can implement exact same system by using these mappings. For instance, if we want to account total boost for validator X, we can fetch `boosted[address(BGTHandler)][Validator_X]` up-to-date variable.

Discussion

Ipetroulakis

Agreed with both solutions. For solution # 1, the entire `BGTHandler` will be refactored. The team should decide before moving on with the rest of the issues/fixes. DemoreXTess will report issues not related with this first finding. The team can confirm them and apply refactoring based on other issues too.

Issue H-2: Missing HONEY reward handling for BGT staker contract

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/5

Summary

BGT boosters earn staking reward from BGT staker contract as HONEY token. However, there is no handling for these reward tokens.

Vulnerability Detail

Whenever we activate a boost for validator, BGT contract stakes that amount to BGT staker contract:

```
delete boostedQueue[user][pubkey];

IBGTStaker(staker).stake(user, amount);

emit ActivateBoost(msg.sender, user, pubkey, amount);
```

BGT staker contracts earn reward as HONEY token (stablecoin of berachain) and distributes that HONEY to stakers proportionally. Current BGT Handler contract only receives reward from validator rewards by off-chain API.

Impact

HONEY will be stuck in BGT staker.

Code Snippet

From BGT Staker:

https://vscode.blockscan.com/80094/0x6371bcfc35d583b0e12fa540a2625f6cde91b3ef

```
    function _stake(address account, uint256 amount) internal virtual {
        if (amount == 0) StakeAmountIsZero.selector.revertWith();

        // set the reward rate after the first stake if there are undistributed
        ↪  rewards
        if (totalSupply == 0 && undistributedRewards > 0) {
            _setRewardRate();
        }

        // update the rewards for the account after `rewardRate` is updated
```

```
@>        _updateReward(account);

        unchecked {
            uint256 totalSupplyBefore = totalSupply; // cache storage read
            uint256 totalSupplyAfter = totalSupplyBefore + amount;
            // `<=` and `<` are equivalent here but the former is cheaper
            if (totalSupplyAfter <= totalSupplyBefore)
            ↪   TotalSupplyOverflow.selector.revertWith();
            totalSupply = totalSupplyAfter;
            // `totalSupply` would have overflowed first because
            ↪   `totalSupplyBefore` >= `_accountInfo[account].balance`
            _accountInfo[account].balance += amount;
        }
        _safeTransferFromStakeToken(msg.sender, amount);
        emit Staked(account, amount);
    }
```

Tool Used

Manual Review

Recommendation

Consider claiming these tokens by `BGTStaker.getReward()` function.

Issue H-3: `executeRedemption` is vulnerable to DoS attack vectors

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/11

Summary

`executeRedemption` function will revert in case of DoS attacks against the BGT Handler contract easily due to heavy gas usage.

Vulnerability Detail

`executeRedemption` has gas heavy operations. It has O(n) time complexity due to for loop. This for loop iterates through redemption requests and increasing the length of this array is very easy for attacker.

```
for (uint256 i = 0; i < redemptionRequests.length; i++) {
            RedemptionRequest storage request = redemptionRequests[i];

            if (request.user == msg.sender) {
                // Check if this request meets the 2000 block delay
                if (block.number >= request.blockNumber + 2000) {
                    // Execute drop for this specific amount
                    _executeDrops(request.bgtQueued);

                    // Burn held rBGT from contract
                    rBgtToken.burn(address(this), request.rBgtAmount);

                    // Execute the redemption - burn BGT for Bera (stored amount)
                    BGT.redeem(msg.sender, request.bgtQueued);
```

Impact

This is a critical vulnerability because redemptions can be DoSed indefinitely with small cost for attacker. Entire BGT Handler can be rendered useless in this case.

Tool Used

Manual Review

Recommendation

It needs lots of change on BGT Handler. I think it should be refactored using `mapping` hash table instead of array. We can reach the same execution logic if we forward expiration system to off-chain. An off-chain keeper can clear expired redemptions by tracking user structs.

Issue H-4: Incorrect `_reduceUserExpiredRedemption` implementation may cause loss of funds

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/14

Summary

`_reduceUserExpiredRedemption` function has four problems:

1. It use incorrect value for reduce amount
2. It checks queued bgt amount in for loop
3. It changes order of user requests because index increase through length
4. `queueToRedeem` variable is not updated after reallocation

These two behaviour may cause loss of funds.

Vulnerability Detail

`_reduceUserExpiredRedemption` function is used for expired redemption requests, it reduce the previous expired redemption request's rBGT amount and it doesn't take rBGT from user in this case.

However, amount of reduced rBGT amount is not correct. It substracts amount of redeem from total expired redemptions but it has to use redeem amount as reduce amount directly.

```
    function _reduceUserExpiredRedemption(address _user, uint256 _amountToRedeem)
    ↪  internal {
        uint256 currentTime = block.timestamp;
        uint256 amountToReduce = getUserExpiredRBgtAmount(_user) - _amountToRedeem;
        RedemptionRequest[] storage userRequests = userRedemptionRequests[_user];

...

            } else {
                // Use only part of expired amount
                rBgtToTake = 0;
                _reduceUserExpiredRedemption(msg.sender, _amount);
            }
```

It's incorrect and attackers can use this bug as attack vector with following scenario:

1. Attacker has 100e18 rBGT expired redemption
2. Attacker requests 99e18 rBGT redemption
3. Code will try to cover it with expired redemption

4. It will calculate reduce amount as 100e18 - 99e18 = 1e18
5. It will substract 1e18 from 100e18 and now expired redemption will be 99e18.
6. 2000 block later he can claim 99e18 BGT (excluding tax)
7. Attacker can also get refund of expired redemption as 99e18.
8. By repeating this attack vector he can drain the contract.

Secondly, in for loop of `_reduceUserExpiredRedemption` it checks bgt amount is bigger than zero which is incorrect check and not necessary. Because all the expired redemptions can be able to used in order to cover the next request.

```
if (currentTime - request.timestamp > redemptionExpiryTime && request.bgtQueued >
↪  0) {
    uint256 toReduce = amountToReduce > request.rBgtAmount ? request.rBgtAmount :
    ↪  amountToReduce;
```

Following scenario can describe the problem:

1. Attacker has 100e18 expired redemption and it's cleaned and reallocated (bgtQueued == 0)
2. Attacker requests a new 99e18 rBGT redemption
3. getUserExpiredRBgtAmount function will return 100e18 because it doesn't check it's cleaned or not
4. However, it won't reduce it because it won't pass the if check in for loop
5. It won't reduce any rBGT from request.
6. Attacker can claim 99e18 rBGT 2000 blocks later
7. He can repeat the steps to drain the contract

Thirdly, it will remove the request if `rBGT` amount reachs to zero, however, this for loop is the only for loop which increase index per loop in the contract which makes it vulnerable because order of requests will be changed and potentially some requests won't be reduced due to edgecase. This situation enables previous attack vector again.

Lastly, reallocation doesn't update `redeemToQueue` variable and it will cause overestimated `redeemToQueue` variable and less unallocated BGT return for other functions.

Example attack path:

1. User requests a 100 rBGT redemption, redeemToQueue will be equal to 100
2. It expires
3. User requests a 99 rBGT redemption
4. It won't substract 99 rBGT from redeem to queue and then it will add 99 rBGT more in `processRedemption` function.

5. Now redeem to queue will be equal to 199 but it should remain same as 100

Impact

It enables some malicious actors to drain the protocol, cause loss of funds.

Tool Used

Manual Review

Recommendation

Consider using amount to redeem directly as reduce amount, remove bgtQueued > 0 check from for loop, make the loop decreasing instead of increasing and lastly update `redeemToQueue` variable in reallocation.

Issue M-1: Bad capital efficiency for `_prepareBgt` function

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/6

Summary

Current system doesn't prioritize high capital efficiency while ordering the actions in `_prepareBgt`.

Vulnerability Detail

`_prepareBgt` function is responsible function to allocate redeem amount of BGT. However, it's using incorrect order of execution. Currently, it's using following sequence:

1. Drop active boosts
2. Cancel boost queue
3. Use other resources

This order of execution is incorrect because it will reduce capital efficiency significantly. We always want to keep active boost amount at high level in order to earn more yield from both BGT Staker and off-chain API.

Impact

This problem will lower potential yield for both off-chain validator rewards and BGT staking rewards (HONEY).

Code Snippet

```
function _prepareBgt(uint256 _amount) internal {
    uint256 remainingToGet = _amount;

    // First, queue drops from active boosts (2000 block delay)
    if (remainingToGet > 0) {
        for (uint256 i = 0; i < validatorList.length && remainingToGet > 0; i++) {
            bytes memory validator = validatorList[i];
            Validator storage validatorInfo = validators[validator];

            if (validatorInfo.activeAmount > 0) {
                uint256 toDrop =
                    remainingToGet > validatorInfo.activeAmount ?
                     ↪  validatorInfo.activeAmount : remainingToGet;
                // Queue drop (2000 block delay)
                BGT.queueDropBoost(validator, uint128(toDrop));

                // Update internal tracking
                validatorInfo.activeAmount -= uint128(toDrop);
                activeValidatorBoosts -= toDrop;
                validatorInfo.queuedDropAmount += uint128(toDrop);
                validatorInfo.queuedDropBlock = uint32(block.number);
                queuedForDrops += toDrop;

                remainingToGet -= toDrop;
            }
        }
    }

    // Second, try to cancel from queued boosts (immediate)
    if (remainingToGet > 0) {
        for (uint256 i = 0; i < validatorList.length && remainingToGet > 0; i++) {
            bytes memory validator = validatorList[i];
            Validator storage validatorInfo = validators[validator];

            if (validatorInfo.queuedAmount > 0) {
                uint256 toCancel =
                    remainingToGet > validatorInfo.queuedAmount ?
                     ↪  validatorInfo.queuedAmount : remainingToGet;

                // Cancel from queue (immediate)
                BGT.cancelBoost(validator, uint128(toCancel));

                // Update internal tracking
                validatorInfo.queuedAmount -= uint128(toCancel);
                queuedValidatorBoosts -= toCancel;

                remainingToGet -= toCancel;
            }
        }
    }

    // Third, try to use unallocated BGT (immediate)
    if (remainingToGet > 0) {
        uint256 unallocated = _getUnallocatedBgtBalance();
        if (unallocated > 0) {
            uint256 toUse = remainingToGet > unallocated ? unallocated :
             ↪  remainingToGet;
            remainingToGet -= toUse;
        }
    }

    require(remainingToGet == 0, "Insufficient boostable BGT");
```

```
}
```

Tool Used

Manual Review

Recommendation

Instead follow this sequence flow:

1. Use available unboosted balance (`unboostedBalanceOf(address(this))`)
2. Use ready to drop validators
3. Cancel boost queue
4. Create drop queue for active boosts

Issue M-2: Hard-coded delay may cause unexpected results for BGT Handler

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/8

Summary

Our delay block number is hard-coded as 2000 blocks. It's not a safe implementation because it can be changed by BGT owner.

Vulnerability Detail

We use 2000 blocks as delay and it's hard-coded in every function. This block delay is correct right now because BGT uses the same block delay too. However, it can be changed on BGT side because it's configured as hard-coded in BGT.

BGT implementation:

https://vscode.blockscan.com/80094/0x656b95E550C07a9ffe548bd4085c72418Ceb1dba

```
function setActivateBoostDelay(uint32 _activateBoostDelay) external onlyOwner {
    if (_activateBoostDelay == 0 || _activateBoostDelay > BOOST_MAX_BLOCK_DELAY) {
        InvalidActivateBoostDelay.selector.revertWith();
    }
    activateBoostDelay = _activateBoostDelay;
    emit ActivateBoostDelayChanged(_activateBoostDelay);
}

/// @inheritdoc IBGT
function setDropBoostDelay(uint32 _dropBoostDelay) external onlyOwner {
    if (_dropBoostDelay == 0 || _dropBoostDelay > BOOST_MAX_BLOCK_DELAY) {
        InvalidDropBoostDelay.selector.revertWith();
    }
    dropBoostDelay = _dropBoostDelay;
    emit DropBoostDelayChanged(_dropBoostDelay);
}
```

Impact

Most probably it will cause DoS for some functions unexpectedly if it's changed to another value.

Code Snippet

```
function activateBoost(bytes calldata _validatorPubkey) external onlyOwner {
↪  Therefore, storage variables will be outdated in this case.
    require(validators[_validatorPubkey].pubkey.length > 0, "Validator does not
    ↪  exist");

    Validator storage validator = validators[_validatorPubkey];
    require(validator.queuedAmount > 0, "No queued boost to activate");
    require(block.number >= validator.queuedBlock + 2000, "Boost activation delay
    ↪  not met");

    // Call BGT contract to activate the boost
    bool success = BGT.activateBoost(address(this), _validatorPubkey);
    require(success, "Failed to activate boost");

    // Update internal tracking
    validator.activeAmount += validator.queuedAmount;
    activeValidatorBoosts += validator.queuedAmount;
    queuedValidatorBoosts -= validator.queuedAmount;
    validator.queuedAmount = 0;

    emit ValidatorBoostActivated(_validatorPubkey, validator.activeAmount);
}
```

Tool Used

Manual Review

Recommendation

Consider fetching this delay amount from BGT implementation.

Discussion

DemoreXTess

It's partially fixed with: https://github.com/roots-fi/roots-core/pull/45

In some places wrong variable is fetched for BGT contract; `activateBoostDelay` and `dropBoostDelay`.

Issue M-3: `_reallocateBgtToBoosting` function may revert in edgecase scenarios

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/9

Summary

`_reallocateBgtToBoosting` function may revert when `block.number == queuedBlock + 2000` because BGT contract doesn't accept == case.

Vulnerability Detail

`_reallocateBgtToBoosting` function accepts == case in if check. However, this case is not accepted in BGT contract.

Our if check:

```
if (validatorInfo.queuedAmount > 0 && block.number >= validatorInfo.queuedBlock +
↪   2000) {
    // Activate all ready boosts (not limited by remainingToAllocate)
    BGT.activateBoost(address(this), validator);
```

BGT if check:

```
function _checkEnoughTimePassed(uint32 blockNumberLast, uint32 blockBufferDelay)
↪   private view returns (bool) {
    unchecked {
        uint32 delta = uint32(block.number) - blockNumberLast;
        if (delta <= blockBufferDelay) return false;
    }
    return true;
}
```

Impact

It will revert in this edgecase scenario, all the functions which use `_reallocateBgtToBoosting` function will face with DoS

Note: `canActivateBoost` and `canExecuteDrop` functions will return incorrect value too.

Tool Used

Manual Review

Recommendation

Do not accept == case.

Issue M-4: Users can immediately get refund if they have at least 1 expired redemption

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/10

Summary

Refunding system incorrectly refund 100% of the user held balance even if he has not expired redemptions

Vulnerability Detail

In `refundExpiredRedemption` function, it checks total expired redemption amount and then if he has bigger expired amount than zero it refunds total held balance to user.

```
uint256 expiredAmount = _getUserExpiredRBgtAmount(msg.sender);
require(expiredAmount > 0, "No expired redemption to refund");

// Return user's held rBGT
uint256 heldRBgt = userHeldRBgt[msg.sender];
if (heldRBgt > 0) {
    rBgtToken.transfer(msg.sender, heldRBgt);
    delete userHeldRBgt[msg.sender];
}
```

This is not correct behaviour because held balance can hold still not expired redemption requests. For istance, user has 2 redemption requests (1 expired and 1 not-expired). System will allow him to get refund for both of them.

Impact

This is not expected behaviour, this scenario will also cause incorrect element existence in requests array because not-expired request won't be removed in this call.

Tool Used

Manual Review

Recommendation

Do not refund 100% held balance, instead refund expired balance.

Issue L-1: Incorrect calculation for unallocated bgt balance cause less unallocated funds estimation

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/7

Vulnerability Detail

`_getUnallocatedBgtBalance` function calculates idle funds by using following method:

```
function _getUnallocatedBgtBalance() internal view returns (uint256) {
    uint256 totalAllocated = queuedForRedemption + activeValidatorBoosts +
     ↪  queuedValidatorBoosts + queuedForDrops;

    return totalBgtBalance > totalAllocated ? totalBgtBalance - totalAllocated : 0;
}
```

This calculation is actually not a bad estimation but not 100% correct. When we increase `queuedForRedemption` by calling `redeem` it already queue some drops through BGT contract in `_prepareBgt` function:

```
    function _prepareBgt(uint256 _amount) internal {
        uint256 remainingToGet = _amount;

        // First, queue drops from active boosts (2000 block delay)
        if (remainingToGet > 0) {
            for (uint256 i = 0; i < validatorList.length && remainingToGet > 0;
             ↪  i++) {
                bytes memory validator = validatorList[i];
                Validator storage validatorInfo = validators[validator];

                if (validatorInfo.activeAmount > 0) {
                    uint256 toDrop =
                        remainingToGet > validatorInfo.activeAmount ?
                         ↪  validatorInfo.activeAmount : remainingToGet;

                    // Queue drop (2000 block delay)
@>                  BGT.queueDropBoost(validator, uint128(toDrop));

                    // Update internal tracking
                    validatorInfo.activeAmount -= uint128(toDrop);
```

Therefore, there is double accounting in this mathematical expression.

Impact

Double accounting will return less amount of unallocated amount and it will reduce capital efficiency in reallocation.

Tool Used

Manual Review

Recommendation

Consider accounting user redemptions differently than queued drops

Issue L-2: Some weird tokens may return unexpected results

Source: https://github.com/sherlock-audit/2025-08-roots-finance-aug-26th/issues/13

Summary

Some ERC20 tokens do not return boolean value for transfers. USDT is the most popular example for this treat.

Following line in `_forwardClaimedTokens` function may fail due to incorrect interface because `IERC20` will expect boolean value for the function calling.

```
require(IERC20(tokenAddress).transfer(incentiveReceiver, claim.amount), "Token
↪  transfer failed");
```

Impact

It will fail in `_forwardClaimedTokens` function.

Tool Used

Manual Review

Recommendation

Use `safeTransfer` which can handle any ERC20 token implementation.

Disclaimers

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

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