BLOCKSEC

Security Audit

Report for bgw-new-staking-evm

Date: September 25, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	bitget
Target	bgw-new-staking-evm

Version History

Version	Date	Description
1.0	September 25, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-03-30 03:43:29+0000

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of bgw-new-staking-evm of bitget.

This project implements a secure staking protocol that allows users to stake BWB tokens in customizable pools. The contract Staking enables pool creation with flexible parameters, stake/unstake operations with signature verification, and proportional reward distribution. It incorporates security features like reentrancy guards and pause functionality while maintaining transparent reward calculations. The system provides a reliable DeFi infrastructure for decentralized staking services through careful state management and validation checks.

Note this audit only focuses on the smart contracts in the following directories/files:

- src/

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
bgw-new-staking-evm	`Version 1`	`3484e6e5ba12a6d73a2f1b7897ac383075c8720c`
	`Version 2`	`bc995da08155972f421b4a32eb346ef6bbbd85ab`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any war-

[1] https://github.com/bitgetwallet/bgw-new-staking-evm

ranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization

* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **six** recommendations and **two** notes.

- High Risk: 1
- Low Risk: 2
- Recommendation: 6
- Note: 2

ID	Severity	Description	Category	Status
1	High	Rewards remain claimable for unstaked orders	Security Issue	Fixed
2	Low	Incorrect update logic of variable `stakeVisible`	Security Issue	Fixed
3	Low	Lack of updating `rewardAmount` in function `unStakeAndClaim()`	Security Issue	Confirmed
4	-	Remove redundant code	Recommendation	Fixed
5	-	Add additional check for `_orderId`	Recommendation	Fixed
6	-	Add additional check for `_pool.poolId`	Recommendation	Fixed
7	-	Gas optimization inside low-level call	Recommendation	Fixed
8	-	Revise code typos	Recommendation	Fixed
9	-	Revise the timestamp comparison in the function `_calcCurrentRewards()`	Recommendation	Fixed
10	-	Potential centralization risks	Note	-
11	-	Ensure sufficient `BWB` token balance for fair reward distribution	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Rewards remain claimable for unstaked orders

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `Staking` implements the function `_claimRewards()` to distribute rewards to users who have active staking positions. However, the function does not validate whether the order referenced by `_orderId` is staked or not, which creates a logical contradiction as rewards should only be accrued for staked funds. This vulnerability allows a user to call the function `claimRewards()` for an order that has been unstaked, potentially claiming rewards for a period after the assets were withdrawn.

```
342   function claimRewards(
343       uint256 _orderId,
344       uint256 _nonce,
```

```
345         bytes calldata _signature
346     ) external whenNotPaused nonReentrant {
347         _claimRewards(_orderId, _nonce, _signature, false, 0);
348     }
349 
350         /**
351      * @notice Internal function to process reward claims
352      * @dev Handles reward calculation, signature verification, and token transfers
353      * @param _orderId ID of the staking order
354      * @param _nonce Unique number to prevent signature reuse
355      * @param _signature Cryptographic signature for verification
356      * @param isFromUnStake Whether this claim is part of an unstake operation
357      * @param unStakeRewardsAmount Pre-calculated reward amount when called from unstake
358      */
359     function _claimRewards(
360         uint256 _orderId,
361         uint256 _nonce,
362         bytes calldata _signature,
363         bool isFromUnStake,
364         uint256 unStakeRewardsAmount
365     ) internal {
366         Order memory order = orderInfo[_orderId];
367         Pool memory pool = poolInfo[order.poolId];
368         Assets memory asset = assetsInfo[msg.sender];
369         if (pool.paused) {
370             revert PoolPaused();
371         }
372 
373         if (order.staker != msg.sender) {
374             revert CallerIsNotStaker();
375         }
376 
377         if (order.claimedRewardAmount >= order.rewardAmount) {
378             revert OrderRewardIsDone();
379         }
380 
381         uint256 currentRewardAmount;
382         if (isFromUnStake && unStakeRewardsAmount > 0) {
383             currentRewardAmount = unStakeRewardsAmount;
384         } else {
385             currentRewardAmount = _calcCurrentRewards(order);
386         }
387 
388         _checkSigner(_nonce, _signature, currentRewardAmount, order.orderId);
389         if (currentRewardAmount == 0) {
390             revert NotRewardsToClaim();
391         }
392 
393         if (pool.claimedRewardAmount + currentRewardAmount > pool.rewardCap) {
394             revert PoolRewardVisibleNotEnough();
395         }
396 
397         poolInfo[pool.poolId].claimedRewardAmount += currentRewardAmount;
```

```
398
399         if (
400             (asset.claimedRewardAmount + currentRewardAmount) >
401             asset.rewardAmount
402         ) {
403             revert AssetRewardNotEnough();
404         }
405
406         assetsInfo[msg.sender].claimedRewardAmount += currentRewardAmount;
407
408         if (
409             (order.claimedRewardAmount + currentRewardAmount) >
410             order.rewardAmount
411         ) {
412             revert OrderRewardNotEnough();
413         }
414         orderInfo[_orderId].claimedRewardAmount += currentRewardAmount;
415         orderInfo[_orderId].lastClaimedTime = block.timestamp;
416
417         IERC20 bwbToken = IERC20(bwb);
418         uint256 beforeThisBalance = bwbToken.balanceOf(address(this));
419         uint256 beforeStakerBalance = bwbToken.balanceOf(msg.sender);
420
421         bwbToken.safeTransfer(msg.sender, currentRewardAmount);
422
423         uint256 afterThisBalance = bwbToken.balanceOf(address(this));
424         uint256 afterStakerBalance = bwbToken.balanceOf(msg.sender);
425
426         if (beforeThisBalance - afterThisBalance != currentRewardAmount) {
427             revert StakingBalanceCheckFailed();
428         }
429         if (afterStakerBalance - beforeStakerBalance != currentRewardAmount) {
430             revert StakerBalanceCheckFailed();
431         }
432
433         emit ClaimRewards(
434             _orderId,
435             currentRewardAmount,
436             block.timestamp,
437             poolInfo[pool.poolId].claimedRewardAmount,
438             assetsInfo[msg.sender].claimedRewardAmount,
439             msg.sender
440         );
441     }
```

Listing 2.1: src/NewStaking.sol

Impact The protocol may transfer reward tokens based on an invalid order state, leading to an unjustified distribution of assets.

Suggestion Check the order's staking status when claiming rewards.

2.1.2 Incorrect update logic of variable `stakeVisible`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `_unStake()`, when a user unstakes before the staking period ends, the contract computes the actual staking time `actualSpendTime` and derives `actualSpendVisible` accordingly. It then takes the difference between the original staked amount and `actualSpendVisible` and adds that difference back to `pool.stakeVisible`. This logic is intended to keep the ratio between pool's staked tokens and reward tokens consistent.

However, due to precision loss, the ratio between the `pool.stakeVisible` and `pool.rewardVisible` of the pool may change, resulting in `DoS` issue. A concrete example is shown below:

1. Suppose the pool's `stakeCap` is `10000` tokens and its `rewardCap` is `6000` tokens. User A stakes `100` tokens and is recorded to receive `60` tokens of reward. The staking period is `30` days, but user A unstakes after `10` days.

2. In this case, user A receives `60 * 10 / 30 = 20` tokens as the earned reward. Suppose there are `10` users who have executed the same logic as user A. The pool's remaining reward becomes `5800` tokens.

3. Meanwhile, the computed actual staked amount for user A becomes `100 * 10 / 30 = 33`. When all the 10 users withdraw the `100` tokens, the pool's remaining available stake tokens is `10000 - 33 * 10 = 9670`, which is recorded as `pool.stakeVisible.`

4. Given `pool.stakeVisible` is `9670`, the distributable reward would be `9670 * (6000 / 10000) = 5802`. However, the actual remaining reward is only `5800.`In this case, when a user stake `9670` tokens, the transaction will revert due to insufficient remaining reward.

```
510   function _unStake(
511       uint256 _orderId,
512       uint256 _nonce,
513       bytes calldata _signature
514   ) internal returns (uint256 unstakedAmount) {
515       Order memory userOrder = orderInfo[_orderId];
516       if (userOrder.unStaked) {
517           revert OrderIsUnStaked();
518       }
519
520       if (msg.sender != userOrder.staker) {
521           revert CallerIsNotStaker();
522       }
523
524       if (poolInfo[userOrder.poolId].paused) {
525           revert PoolPaused();
526       }
527
528       unstakedAmount = userOrder.stakeAmount;
529
530       _checkSigner(_nonce, _signature, unstakedAmount, _orderId);
531
532       orderInfo[_orderId].unStaked = true;
```

```
533         assetsInfo[msg.sender].stakedAmount -= unstakedAmount;
534
535         // allow unstake at anytime, refund stake visiable & reward visible
536         if (block.timestamp < userOrder.unStakeTime) {
537             // compute actual unspend
538             uint256 actualSpendTime = (block.timestamp - userOrder.createTime) / SECONDS_ONE_DAY;
539             uint256 actualSpendVisible = actualSpendTime * userOrder.stakeAmount / poolInfo[
                    userOrder.poolId].durationDays;
540
541             poolInfo[userOrder.poolId].stakeVisible += userOrder.stakeAmount - actualSpendVisible;
542             poolInfo[userOrder.poolId].rewardVisible += userOrder.rewardAmount - userOrder.
                    claimedRewardAmount;
543         }
544
545         IERC20 bwbToken = IERC20(bwb);
546
547         uint256 beforeThisBalance = bwbToken.balanceOf(address(this));
548         uint256 beforeStakerBalance = bwbToken.balanceOf(msg.sender);
549
550         bwbToken.safeTransfer(msg.sender, unstakedAmount);
551
552         uint256 afterThisBalance = bwbToken.balanceOf(address(this));
553         uint256 afterStakerBalance = bwbToken.balanceOf(msg.sender);
554
555         if (beforeThisBalance - afterThisBalance != unstakedAmount) {
556             revert StakingBalanceCheckFailed();
557         }
558
559         if (afterStakerBalance - beforeStakerBalance != unstakedAmount) {
560             revert StakerBalanceCheckFailed();
561         }
562     }
```

Listing 2.2: src/NewStaking.sol

Impact Users may not execute stake operations.

Suggestion Modify the calculation logic to calculate the variable `stakeVisible` based on the variable `rewardVisible`.

Clarification from BlockSec The project fix the issue by modifying the calculation logic to calculate the variable `stakeVisible` based on the variable `rewardVisible`. In this case, the contract `NewStaking` may end up with residual reward tokens, which the project can retrieve using the `rescueERC20()` function.

2.1.3 Lack of updating `rewardAmount` in function `unStakeAndClaim()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The contract maintains user asset information in the `Assets` struct, which tracks `stakedAmount`, `rewardAmount` (total estimated rewards), and `claimedRewardAmount` for each user.

However, when a user invokes function `unStakeAndClaim()` before the staking period ends, the user's `rewardAmount` field is not updated. This leads to an incorrect recorded reward amount in the user's asset record.

```
399       if (
400           (asset.claimedRewardAmount + currentRewardAmount) >
401           asset.rewardAmount
402       ) {
403           revert AssetRewardNotEnough();
404       }
405
406       assetsInfo[msg.sender].claimedRewardAmount += currentRewardAmount;
```

Listing 2.3: src/NewStaking.sol

```
533       assetsInfo[msg.sender].stakedAmount -= unstakedAmount;
534
535       // allow unstake at anytime, refund stake visiable & reward visible
536       if (block.timestamp < userOrder.unStakeTime) {
537           // compute actual unspend
538           uint256 actualSpendTime = (block.timestamp - userOrder.createTime) / SECONDS_ONE_DAY;
539           uint256 actualSpendVisible = actualSpendTime * userOrder.stakeAmount / poolInfo[
                  userOrder.poolId].durationDays;
540
541           poolInfo[userOrder.poolId].stakeVisible += userOrder.stakeAmount - actualSpendVisible;
542           poolInfo[userOrder.poolId].rewardVisible += userOrder.rewardAmount - userOrder.
                  claimedRewardAmount;
543       }
```

Listing 2.4: src/NewStaking.sol

Impact Frontend interfaces relying on the struct `Assets` will display inaccurate reward amounts, leading to user confusion and a degraded user experience.

Suggestion Revise the code accordingly.

Feedback from the project The project specifies that the `rewardAmount` field in the `Asset` struct will not be utilized for any off-chain program calculations.

2.2 Recommendation

2.2.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `StakingErrors`, the error `TooEarlyToClaim()` is declared but never used. It is recommended to either remove the unused error declaration.

```
29    error TooEarlyToUnStake();
```

Listing 2.5: src/StakingErrors.sol

Suggestion Remove the redundant code.

2.2.2 Add additional check for `_orderId`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Staking`, the functions `getOrderRewardsAmount()` and `getRewardsDetails()` do not check whether an order corresponding to `_orderId` exists, which may result in an expected behavior.

```
565     * @notice Gets the current available rewards for a specific staking order
566     * @dev Public view function that calls _calcCurrentRewards
567     * @param _orderId ID of the staking order
568     * @return The amount of rewards currently available to claim
569     */
570    function getOrderRewardsAmount(
571        uint256 _orderId
572    ) public view returns (uint256) {
573        return _calcCurrentRewards(orderInfo[_orderId]);
574    }
```

Listing 2.6: src/NewStaking.sol

```
599    function getRewardsDetails(uint256 _orderId) view external returns (uint256,uint256,uint256,
           uint256,uint256,uint256,uint256,uint256,uint256) {
600        Order memory _order = orderInfo[_orderId];
601        Pool memory _pool = poolInfo[_order.poolId];
602        uint256 currentRewardAmount = _calcCurrentRewards(_order);
603        return (_order.poolId, _pool.rewardCap, _pool.duration, _order.createTime, _order.
               unStakeTime, _order.lastClaimedTime, _order.rewardAmount, _order.claimedRewardAmount,
               currentRewardAmount);
604    }
```

Listing 2.7: src/NewStaking.sol

Suggestion Revise the code accordingly.

2.2.3 Add additional check for `_pool.poolId`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Staking`, the function `updatePool()` does not check whether a pool corresponding to `_pool.poolId` exists, which may result in updating a non-exist pool.

```
134    function updatePool(Pool memory _pool) external onlyAdmin whenNotPaused {
135        if (block.timestamp >= poolInfo[_pool.poolId].startTime) {
136            revert PoolIsStarted();
137        }
138
139        if (
140            _pool.stakeCap != _pool.stakeVisible ||
141            _pool.rewardCap != _pool.rewardVisible
142        ) {
```

```
143             revert PoolCapAndVisibleMismatched();
144         }
145
146         if ( _pool.rewardRate != (_pool.rewardCap * DECI) / _pool.stakeCap) {
147             revert PoolRewardRateMismatched();
148         }
149
150         if (_pool.claimedRewardAmount != 0) {
151             revert PoolInitClaimedRewardIsZero();
152         }
153
154         _checkPoolAvailable(
155             _pool.stakeCap,
156             _pool.rewardCap,
157             _pool.startTime,
158             _pool.endTime,
159             _pool.duration,
160             _pool.durationDays
161         );
162         poolInfo[_pool.poolId] = _pool;
163
164         emit CreateOrUpdatePool(
165             _pool.poolId,
166             _pool.stakeCap,
167             _pool.rewardCap,
168             _pool.rewardRate,
169             _pool.stakeVisible,
170             _pool.rewardVisible,
171             _pool.startTime,
172             _pool.endTime,
173             _pool.duration,
174             _pool.durationDays,
175             _pool.paused
176         );
177     }
```

Listing 2.8: src/NewStaking.sol

Suggestion Revise the code accordingly.

2.2.4 Gas optimization inside low-level call

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `rescueETH()` utilizes a low-level call with an empty data payload to send native `Ether`. While this is functional, it is not gas-efficient since allocating a new empty bytes array incurs additional gas costs. A more gas-efficient approach is to use an empty string (i.e., `""`) as the data payload, which achieves the same result without the overhead of dynamic memory allocation.

```
108         (bool success, ) = safe.call{value: amount}(new bytes(0));
```

Listing 2.9: src/StakingBase.sol

Suggestion Revise the code accordingly.

2.2.5 Revise code typos

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several code typos, which affect the readability of the code and is recommended to revise them. Specifically:

1. The error name `OrderRewardUnavailabled` is misspelled.

```
33    error OrderRewardUnavailabled();
```

Listing 2.10: src/StakingErrors.sol

```
459           revert OrderRewardUnavailabled();
```

Listing 2.11: src/NewStaking.sol

2. The event `SetBWB` is defined with a parameter `admin` but it is used to log the `BWB` token address.

```
5    event SetBWB(address admin);
```

Listing 2.12: src/StakingEvents.sol

```
54    function _setBWB(address _bwb) internal {
55        _checkZero(_bwb);
56        _checkEqual(_bwb, bwb);
57        bwb = _bwb;
58        emit SetBWB(_bwb);
59    }
```

Listing 2.13: src/StakingBase.sol

Suggestion Revise code typos.

2.2.6 Revise the timestamp comparison in the function `_calcCurrentRewards()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The condition `block.timestamp > _order.unStakeTime` in the function `_calcCurrentRewards()` uses a greater-than comparison, which leads to higher gas cost when the timestamps are equal. It is recommended to use greater-than-or-equal comparison instead to optimize gas consumption.

```
452        if (block.timestamp > _order.unStakeTime) {
```

Listing 2.14: src/NewStaking.sol

Suggestion Revise the timestamp comparison in the function `_calcCurrentRewards()`.

2.3 Note

2.3.1 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., `admin`, `operator`) can conduct sensitive operations, which introduces potential centralization risks. For example, the `admin` can update pool configurations, directly affecting users' reward calculations, and the `operator` can rescue the BWB tokens in the contract `Staking`. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.2 Ensure sufficient `BWB` token balance for fair reward distribution

Introduced by Version 1

Description The current contract code does not include validation to ensure sufficient `BWB` token balance for reward distribution before processing staking operations. To guarantee fair and uninterrupted reward distribution, the project must proactively maintain an adequate `BWB` token balance in the contract to fully cover all promised rewards.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS