BLOCKSEC

Security Audit Report for DGAIStaking

Date: June 25, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DgridAI
Target	DGAIStaking

Version History

Version	Date	Description
1.0	June 25, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-19 05:18:52+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

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

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of DGAIStaking of DgridAI.

DGAIStaking is a staking and reward distribution protocol for the DGAI ecosystem. The protocol allows users to stake DGAI through self-operated nodes, delegate DGAI to existing nodes, or participate in a dedicated LLM staking pool. Rewards are distributed across multiple reward categories, including node, team, and LLM incentives, and can be claimed by eligible participants based on their staking positions. The protocol also supports unstaking with a cooldown period before principal withdrawal. Administrative functions include reward allocation management, staking parameter configuration, pause controls, and treasury-related fund management.

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

- contracts/DGAIStaking.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the Target 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
DGRID	`Version 1`	`8b04c8f352ae655a1cd0eddce66e7d4d56c5db82`
	`Version 2`	`4d7a58cad44156d654cf57d763907945430fb903`

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.

[1] https://github.com/dgridai/DGRID

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 warranties on discovering all security issues of the Target, 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 the Target.

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 the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target 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 **seven** potential security issues. Besides, we have **two** recommendations and **three** notes.

- High Risk: 2
- Medium Risk: 2
- Low Risk: 3
- Recommendation: 2
- Note: 3

ID	Severity	Description	Category	Status
1	High	Improper team reward accounting after `dev` address reassignment	Security Issue	Fixed
2	High	Unreserved user principal in reward distribution mechanism	Security Issue	Fixed
3	Medium	Unaccounted reward remainders due to precision loss	Security Issue	Fixed
4	Medium	Unsettled team rewards accruals during `dev` reassignment	Security Issue	Fixed
5	Low	Unaccounted reward liabilities in function `emergencyWithdraw()`	Security Issue	Confirmed
6	Low	Lack of node deregistration mechanism	Security Issue	Confirmed
7	Low	Inconsistent emission reporting for disabled node group	Security Issue	Fixed
8	-	Remove redundant `dev` checks	Recommendation	Fixed
9	-	Emit events for administrative updates	Recommendation	Fixed
10	-	Potential centralization risks	Note	-
11	-	Reward funding dependency	Note	-
12	-	Proxy initialization atomicity assumption	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Improper team reward accounting after `dev` address reassignment

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Team rewards are tracked using the current `dev` address as the accounting identity. The accrued reward state is maintained through `teamRewardDebt` and `teamUnpaid`, which are indexed by address.

However, when the owner invokes function `setDev()` to assign a new `dev` address, the reward accounting state is not migrated or initialized for the new address. As a result, the newly assigned `dev` starts with a zero `teamRewardDebt` regardless of the rewards that have already been accounted for under the previous `dev`.

When the new `dev` subsequently claims team rewards, the pending reward calculation treats the address as if it had participated since the beginning of the reward program, allowing it to claim historical team rewards that were accrued before the reassignment.

```
799   function setDev(address _dev) external onlyOwner {
800       dev = _dev;
801   }
```

Listing 2.1: contracts/DGAIStaking.sol

```
995   function _resetTeamDebt(address _user) internal {
996       if (_user != dev) {
997           return;
998       }
999       teamRewardDebt[_user] =
1000          (TEAM_SHARE * accRewardPerShares[TEAM_GROUP_ID]) /
1001          ACC_PRECISION;
1002  }
```

Listing 2.2: contracts/DGAIStaking.sol

Impact A newly assigned `dev` address may claim team rewards that were accrued prior to the reassignment. This results in incorrect team reward distribution and may lead to double allocation of historical team rewards between the previous and current `dev` addresses.

Suggestion Update the new dev address's `teamRewardDebt` to the current accumulated reward value when invoking function `setDev()`.

2.1.2 Unreserved user principal in reward distribution mechanism

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The protocol distributes rewards through several reward claim functions(e.g., function `claim()`), allowing eligible users to claim accrued rewards. User deposits are tracked through `totalPrincipal`, while rewards are distributed from the `DGAI` balance held by the contract.

However, the reward claim logic does not verify that sufficient reward funds exist beyond the tracked user principal before transferring rewards. Instead, rewards are paid directly from the contract balance based solely on the accrued reward accounting. As a result, if the available reward balance becomes insufficient, reward payouts may be funded using staked principal or funds required to satisfy future withdrawals.

Consequently, reward distributions may reduce the amount of `DGAI` available for principal redemption and compromise the solvency of the staking pool.

```
479   function claim(address _node) external nodeExists(_node) nonReentrant {
480       require(
481           block.timestamp >=
482               lastTimeClaimNode[_node][msg.sender] +
483                   (uint256(coolingClaimDay) * 1 days),
```

```
484             "claim cooling"
485         );
486         updateNodePool();
487         _accrueUnpaid(_node, msg.sender);
488
489         uint256 amount = userUnpaid[_node][msg.sender];
490         require(amount > 0, "no rewards , maybe stake activity not started");
491
492         userUnpaid[_node][msg.sender] = 0;
493         lastTimeClaimNode[_node][msg.sender] = block.timestamp;
494         DGAI.safeTransfer(msg.sender, amount);
495
496         emit Claim(_node, msg.sender, amount);
497     }
```

Listing 2.3: contracts/DGAIStaking.sol

Impact Reward claims may consume staked principal, resulting in insufficient funds for future withdrawals. In severe cases, the staking pool may become insolvent and users may be unable to fully withdraw their deposited `DGAI`.

Suggestion Ensure that reward distributions are restricted to funds exceeding the tracked user principal and cannot consume staked deposits.

2.1.3 Unaccounted reward remainders due to precision loss

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `_updateGroup()` is responsible for updating a group's reward accounting and distributing newly accrued rewards to stakers through the accumulated reward-per-share mechanism.

However, small reward accruals may not be reflected in `accRewardPerShares` due to integer precision limitations. Despite this, the function still advances `group.lastRewardTime` and treats the elapsed period as fully processed.

As a result, the rewards accrued during that period are omitted from the group's reward accounting and cannot be recovered by subsequent updates. Since functions `updateNodePool()`, `updateTeamPool()`, and `updateLlmPool()` can be invoked frequently, these unaccounted rewards may accumulate over time, causing the total distributed rewards to be lower than intended.

```
841         if (reward > 0) {
842             accRewardPerShares[_groupId] +=
843                 (reward * ACC_PRECISION) /
844                 group.totalStaked;
845         }
```

Listing 2.4: contracts/DGAIStaking.sol

Impact A portion of accrued rewards may become permanently unclaimable due to repeated precision loss during pool updates. Over time, stakers may receive fewer rewards than intended by the configured reward schedule.

Suggestion Preserve precision-loss remainders across updates so that omitted rewards can be incorporated into future reward calculations.

2.1.4 Unsettled team rewards accruals during `dev` reassignment

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Team rewards are accrued to the current `dev` address and become claimable through the team reward accounting mechanism.

However, the function `setDev()` directly replaces the current `dev` address without first settling the team rewards accrued under the previous `dev`. Once the reassignment is completed, the previous `dev` is no longer authorized to claim team rewards, while the accrued rewards remain associated with the previous reward accounting state.

As a result, team rewards accrued before the reassignment may remain unclaimed after the `dev` address is updated.

```
799    function setDev(address _dev) external onlyOwner {
800        dev = _dev;
801    }
```

Listing 2.5: contracts/DGAIStaking.sol

Impact Team rewards accrued prior to a `dev` reassignment may become inaccessible, resulting in a permanent loss of rewards intended for the previous `dev`.

Suggestion Settle the previous `dev`'s pending team rewards before updating the `dev` address.

2.1.5 Unaccounted reward liabilities in function `emergencyWithdraw()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The function `emergencyWithdraw()` allows the owner to withdraw `DGAI` held by the contract in excess of `totalPrincipal`, which represents the total amount of user staked principal.

However, the contract may also hold `DGAI` that is already owed to users as accrued but unclaimed rewards. These reward obligations are tracked through variables such as `userUnpaid`, `nodeCommissionUnpaid`, `llmUserUnpaid`, and `teamUnpaid`, but are not considered when calculating the withdrawable amount.

As a result, funds reserved for pending reward claims may be treated as withdrawable surplus and transferred out of the contract through function `emergencyWithdraw()`.

```
1006    function emergencyWithdraw(
1007        address _target,
1008        uint256 _amount
1009    ) external onlyOwner {
1010        require(_target != address(0), "target is zero");
1011        require(_amount > 0, "amount is zero");
1012
1013        uint256 balance = DGAI.balanceOf(address(this));
1014        uint256 withdrawable = balance > totalPrincipal
1015            ? balance - totalPrincipal
1016            : 0;
1017        require(_amount <= withdrawable, "exceeds withdrawable surplus");
1018
1019        DGAI.safeTransfer(_target, _amount);
1020        emit EmergencyWithdraw(address(DGAI), _target, _amount);
1021    }
```

Listing 2.6: contracts/DGAIStaking.sol

Impact Accrued but unclaimed rewards may become partially or fully unclaimable if funds reserved for pending reward distributions are withdrawn through function `emergencyWithdraw()`.

Suggestion Exclude all accrued but unclaimed rewards from the withdrawable amount calculated by function `emergencyWithdraw()`.

Feedback from the project In production, the `owner` is expected to be controlled by a multisig or timelock contract, which prevents this issue from occurring.

2.1.6 Lack of node deregistration mechanism

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Node operators are required to maintain a minimum self-staked amount when invoking function `unstake()`. As a result, a node operator cannot reduce its self-stake below `minNodeSelfStakeAmount`.

However, the protocol does not provide any mechanism to deregister, deactivate, or remove an existing node. Consequently, once a node is created, the operator has no supported way to fully exit the node role and withdraw all self-staked funds.

Furthermore, the issue is not limited to configurations where `minNodeSelfStakeAmount` is greater than zero. Even when the minimum self-stake requirement is configured as zero, a node remains active after its self-stake is fully withdrawn, and other users may continue delegating funds to that node. As a result, the protocol does not define a complete node lifecycle or a clear exit path for node operators.

```
385        if (msg.sender == _node) {
386            uint256 remaining = userAmount[_node][msg.sender] - _amount;
387            require(
388                remaining >= minNodeSelfStakeAmount,
```

```
389             "self stake below minimum"
390         );
391     }
```

Listing 2.7: contracts/DGAIStaking.sol

Impact Node operators may be unable to fully exit the node role. Depending on the protocol configuration, a portion of self-staked funds may remain locked, or inactive nodes may continue accepting user delegations despite having no operator stake remaining.

Suggestion Introduce an explicit node deregistration mechanism and define the expected behavior of existing delegations when a node exits the protocol.

Clarification The protocol has reserved the `nodeStatus` field in the node data structure, which can support future node deregistration or other lifecycle controls through contract upgrades without changing the existing storage layout.

2.1.7 Inconsistent emission reporting for disabled node group

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `getAnnualStakingEmission()` is intended to provide an estimate of the annual reward emission for each staking group.

However, when the node reward mode is configured as `FixedRate`, the function returns the annual emission directly based on the configured reward rate without considering whether the node group is currently enabled. In contrast, the actual reward distribution logic in function `_updateGroup()` exits early when `group.enabled` is false, preventing any rewards from being accrued or distributed.

As a result, when the node group is disabled and the reward mode is `FixedRate`, the protocol distributes no node rewards while function `getAnnualStakingEmission()` continues to report a positive annual emission. This creates an inconsistency between the reported emission data and the actual reward behavior of the protocol.

```
682   function getAnnualStakingEmission(
683       uint256 _groupId
684   ) external view returns (uint256 released) {
685       require(_groupId < groupInfos.length, "invalid group");
686
687       if (_groupId == NODE_GROUP_ID) {
688           if (nodeRewardMode == NodeStakeMode.FixedRate) {
689               return
690                   (totalStakedAmount * uint256(annualRewardRate)) /
691                   BPS_DENOMINATOR;
692           }
693           if (nodeRewardMode == NodeStakeMode.NoReward) {
694               return 0;
695           }
696       }
```

```
697
698         if (!groupInfos[_groupId].enabled) {
699             return 0;
700         }
701
702         return uint256(groupInfos[_groupId].perSecondReward) * 365 days;
703     }
```

Listing 2.8: contracts/DGAIStaking.sol

Impact Frontends, dashboards, and off-chain integrations relying on function `getAnnualStakingEmission()` can display inaccurate reward estimates for disabled node groups.

Suggestion Ensure that function `getAnnualStakingEmission()` applies the same group enablement checks as the reward distribution logic.

2.2 Recommendation

2.2.1 Remove redundant `dev` checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The functions `_accrueTeamUnpaid()` and `_resetTeamDebt()` contain checks that return early when `_user != dev`.

However, function `_accrueTeamUnpaid()` is only invoked using the current `dev` address, and function `_resetTeamDebt()` is only reachable through function `_accrueTeamUnpaid()`. As a result, the `_user != dev` condition cannot be satisfied under the current call flow.

These checks therefore do not provide any additional protection and introduce unnecessary conditional logic.

```
962         if (_user != dev) {
963             return;
964         }
```

Listing 2.9: contracts/DGAIStaking.sol

```
996         if (_user != dev) {
997             return;
998         }
```

Listing 2.10: contracts/DGAIStaking.sol

Suggestion Remove the redundant `_user != dev` checks.

2.2.2 Emit events for administrative updates

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The functions `setDev()`, `setMinLlmStakeAmount()`, `setMinDelegatorStakeAmount()`, and `setMinNodeSelfStakeAmount()` modify protocol configuration without emitting events.

These functions update privileged roles and staking-related parameters that may affect protocol operation and user behavior. However, unlike other administrative setters in the contract, these changes are not recorded through event emissions.

```
799   function setDev(address _dev) external onlyOwner {
800       dev = _dev;
801   }
```

Listing 2.11: contracts/DGAIStaking.sol

```
705   function setMinLlmStakeAmount(uint256 _minAmount) external onlyOwner {
706       minLlmStakeAmount = _minAmount;
707   }
708
709   function setMinDelegatorStakeAmount(uint256 _minAmount) external onlyOwner {
710       minDelegatorStakeAmount = _minAmount;
711   }
712
713   function setMinNodeSelfStakeAmount(uint256 _minAmount) external onlyOwner {
714       minNodeSelfStakeAmount = _minAmount;
715   }
```

Listing 2.12: contracts/DGAIStaking.sol

Suggestion Emit events for functions mentioned above to improve transparency and off-chain observability.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `Owner`) can conduct sensitive operations, which introduces potential centralization risks. For example, the Owner can arbitrarily increase the claim cooldown period via function `setCoolClaimDay()`, preventing users from claiming accrued rewards, and arbitrarily increase the unstake cooldown period via function `setCoolingUnstakeDay()`, locking users' staked funds and preventing withdrawals for an arbitrarily long period of time. 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 Reward funding dependency

Introduced by `Version 1`

Description The contract records reward accruals through on-chain accounting but does not enforce that sufficient `DGAI` is available to satisfy future reward claims. As a result, reward liabilities may continue to accumulate regardless of the contract's actual token balance.

The ability of users to successfully claim rewards therefore depends on the project maintaining adequate `DGAI` funding within the contract. If the available balance becomes insufficient, reward claims may fail despite the corresponding rewards having already been accrued on-chain.

2.3.3 Proxy initialization atomicity assumption

Introduced by `Version 1`

Description To prevent potential front-running attacks, it is recommended that proxy deployment and implementation binding be executed atomically within a single transaction. If these operations are performed separately, malicious actors could exploit the time window between deployment and binding to front-run the implementation binding transaction. An attacker could potentially bind their own malicious implementation to the newly deployed proxy contract, gaining unauthorized control over the protocol's upgrade mechanism and user funds. This race condition poses significant security risks including complete protocol compromise, fund theft, and unauthorized access to privileged functions.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS