Security Audit Report for Q101 Token Smart Contracts

Date: January 4, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction 1

1.1 About Target Contracts . . . 1

1.2 Disclaimer . . . 1

1.3 Procedure of Auditing . . . 2

1.3.1 Security Issues . . . 2

1.3.2 Additional Recommendation . . . 3

1.4 Security Model . . . 3

Chapter 2 Findings 4

2.1 Security Issue . . . 4

2.1.1 Premature vesting release due to rounding down . . . 4

2.1.2 Potential replay risks due to lack of domain separation . . . 6

2.1.3 Potential DoS on airdrop claims . . . 6

2.1.4 Lack of `voucherId` invalidation for failed reveal attempts . . . 7

2.2 Recommendation . . . 9

2.2.1 Validate configured `startTime` in function `configureAirdrop()` . . . 9

2.2.2 Remove redundant code . . . 10

2.2.3 Fix conflicts in documentation . . . 10

2.2.4 Optimize ownership grant logic . . . 11

2.3 Note . . . 11

2.3.1 Ensure secure generation of vouchers . . . 11

2.3.2 Potential centralization risks . . . 12

2.3.3 Proxy deployment and implementation binding should be atomic . . . 12

2.3.4 Ensure sufficient token balances in contract `Q101AirdropVesting` . . . 12

2.3.5 Security of Gelato integration . . . 13

2.3.6 Merkle tree modification should only add new vouchers . . . 13

Report Manifest

Item	Description
Client	Open Quest Academy
Target	Q101 Token Smart Contracts

Version History

Version	Date	Description
1.0	January 4, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-01-06 08:35:32+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 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 Q101 Token Smart Contracts of Open Quest Academy.

The project consists of two upgradeable contracts, `Q101Token` and `Q101AirdropVesting`. Contract `Q101Token` is an ERC-20 token with a fixed supply of 1 billion tokens, featuring emergency pause functionality and UUPS upgradeability. The ownership and initially minted total supply are controlled by a Gnosis Safe multi-signature address. Contract `Q101AirdropVesting` serves as the airdrop and vesting contract for `Q101Token`. It features a commit-reveal mechanism to prevent front-running during airdrop claims, Merkle proof verification for eligibility, and a three-stage token release model. The contract also supports gasless transactions through Gelato Relay integration.

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
q101-coin-smart-contract	Version 1	2eb2bbd02d0a3da0c00151eaa656b0c5cc0630e3
	Version 2	d4a301eb606beb64f153dc269bc0181488e2df97

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/Open-Quest-Academy/q101-coin-smart-contract

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 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 **four** potential security issues. Besides, we have **four** recommendations and **six** notes.

- Medium Risk: 3
- Low Risk: 1
- Recommendation: 4
- Note: 6

ID	Severity	Description	Category	Status
1	Medium	Premature vesting release due to rounding down	Security Issue	Fixed
2	Medium	Potential replay risks due to lack of domain separation	Security Issue	Confirmed
3	Medium	Potential DoS on airdrop claims	Security Issue	Fixed
4	Low	Lack of `voucherId` invalidation for failed reveal attempts	Security Issue	Fixed
5	-	Validate configured `startTime` in function `configureAirdrop()`	Recommendation	Confirmed
6	-	Remove redundant code	Recommendation	Fixed
7	-	Fix conflicts in documentation	Recommendation	Fixed
8	-	Optimize ownership grant logic	Recommendation	Fixed
9	-	Ensure secure generation of vouchers	Note	-
10	-	Potential centralization risks	Note	-
11	-	Proxy deployment and implementation binding should be atomic	Note	-
12	-	Ensure sufficient token balances in contract `Q101AirdropVesting`	Note	-
13	-	Security of Gelato integration	Note	-
14	-	Merkle tree modification should only add new vouchers	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Premature vesting release due to rounding down

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Q101AirdropVesting`, the function `_calculateLinearVested()` calculates the vested token amount based on global duration and frequency parameters. However, the calculation logic is incorrect when the vesting frequency is set to `PER_DAY` or `PER_MONTH` because the total number of periods is rounded down during integer division. As a result, this design flaw causes the entire token allocation to be released before the actual vesting duration concludes.

```
function _calculateLinearVested(
    uint256 vestingBase,
    uint256 vestingElapsed,
    uint256 duration
) internal view returns (uint256) {
    // If vesting period completed, return all
    if (vestingElapsed >= duration) {
        return vestingBase;
    }

    // Calculate based on frequency mode
    if (vestingFrequency == VestingFrequency.PER_SECOND) {
        // Per second: most precise
        return (vestingBase * vestingElapsed) / duration;
    }
    else if (vestingFrequency == VestingFrequency.PER_DAY) {
        // Per day: vests once per day
        uint256 totalDays = duration / 1 days;
        uint256 elapsedDays = vestingElapsed / 1 days;

        if (elapsedDays >= totalDays) {
            return vestingBase;
        }
        return (vestingBase * elapsedDays) / totalDays;
    }
    else if (vestingFrequency == VestingFrequency.PER_MONTH) {
        // Per month: vests once per 30 days
        uint256 totalMonths = duration / 30 days;
        uint256 elapsedMonths = vestingElapsed / 30 days;

        if (elapsedMonths >= totalMonths) {
            return vestingBase;
        }
        return (vestingBase * elapsedMonths) / totalMonths;
    }

    return 0;
}
```

Listing 2.1: src/Q101AirdropVesting.sol

Impact The rounding down error leads to the premature release of tokens before the vesting duration officially ends.

Suggestion Add a check in the function `configureAirdrop()` to ensure that the vesting duration is an exact multiple of the vesting frequency.

2.1.2 Potential replay risks due to lack of domain separation

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the contract `Q101AirdropVesting`, the Merkle leaf is calculated using `keccak256(voucherId, amount)`, which lacks domain separation elements such as the chain ID or the contract address. Consequently, if the same Merkle root is reused across different contract instances on the same chain or across different chains, a valid proof for one contract can be replayed on another. This risk is particularly relevant as the project intends to deploy two separate vesting contracts on the BSC network. If a unified Merkle tree is used for both, it will enable cross-contract replay attacks.

```
        // 6. Calculate leaf hash
        bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(voucherId, amount))));
```

Listing 2.2: src/Q101AirdropVesting.sol

Impact Users can double-claim tokens by replaying valid Merkle proofs across different contract instances or chains, leading to a loss of funds for the project.

Suggestion Include `block.chainid` and `address(this)` in the Merkle leaf calculation to ensure domain separation.

Feedback from the project The project confirmed that the Merkle Root is generated off-chain using a new set of high-entropy random vouchers, ensuring that each airdrop's Merkle Root differs from previous ones.

2.1.3 Potential DoS on airdrop claims

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Q101AirdropVesting`, function `_createAndWithdrawImmediatelyReleasable()` verifies that the contract balance covers the total allocation `amount`. However, the project employs a staged funding model where contract liquidity is provided incrementally over time. This verification is incompatible with the funding mechanism. Consequently, even if the contract funds are sufficient for immediate release but are less than the total allocation, the verification would fail. This design would result in a denial-of-service (DoS) on legitimate claims.

```
    function _createAndWithdrawImmediatelyReleasable(address user, uint256 amount) internal {
        require(token.balanceOf(address(this)) >= amount, "Contract: Insufficient tokens");

        // Calculate immediate release amount (Stage 1)
        uint256 immediateAmount = (amount * immediateReleaseRatio) / RATIO_PRECISION;

        // Create vesting schedule
```

```
    vestingSchedules[user] = VestingSchedule({
        startTime: startTime,
        duration: uint64(vestingDuration),
        totalAmount: amount,
        immediateAmount: immediateAmount,
        releasedAmount: 0,
        lastWithdrawTime: uint64(block.timestamp)
    });

    emit VestingScheduleCreated(user, amount, startTime);

    // Calculate total releasable amount (includes immediate + vested)
    uint256 totalClaimAmount = _calculateReleasable(user);

    // Update released amount
    vestingSchedules[user].releasedAmount = totalClaimAmount;

    // Transfer tokens
    if (totalClaimAmount > 0) {
        require(token.transfer(user, totalClaimAmount), "Transfer failed");
    }
}
```

Listing 2.3: src/Q101AirdropVesting.sol

Impact This design results in a DoS for legitimate claims, leading to an inability for users to initialize their vesting schedules.

Suggestion Revise the logic accordingly.

2.1.4 Lack of `voucherId` invalidation for failed reveal attempts

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Q101AirdropVesting`, function `reveal()` fails to mark a valid `voucherId` as used when the function execution fails due to timing constraints or other execution failures.

For instance, if users submit the reveal request with the valid `voucherId` and `amount` at 254 blocks after commitment, their transaction may be minted after several blocks, exceeding the allowed delay range. In this scenario, the transaction reverts while the committed information is still revealed on-chain. With revealed information, malicious actors can construct their own `commitHash` to commit and claim airdrops intended for users.

```
/**
 * @notice Reveal the committed data and execute claim (gasless via Gelato Relay)
 * @dev Second step of commit-reveal mechanism, creates vesting schedule and releases tokens
 *      Uses ERC2771 to get real user address from _msgSender()
 * @param voucherId Unique voucher ID
 * @param amount Total allocation amount (in wei)
 * @param salt Random salt used in commitment
 * @param merkleProof Merkle proof for verification
```

```
     */
    function reveal(
        bytes32 voucherId,
        uint256 amount,
        bytes32 salt,
        bytes32[] calldata merkleProof
    ) external whenNotPaused {
        // 0. Get real user address via ERC2771
        address user = _msgSender();

        // 1. Check that merkleRoot has been set
        require(merkleRoot != bytes32(0), "Airdrop not started: merkle root not set");

        // 2. Reconstruct commitment hash
        bytes32 commitHash = keccak256(abi.encode(voucherId, user, amount, salt));

        // 3. Verify commitment exists
        Commitment storage commitment = commitments[commitHash];
        require(commitment.blockNumber > 0, "Reveal: No commitment found");
        require(!commitment.revealed, "Reveal: Already revealed");
        require(commitment.committer == user, "Reveal: Wrong committer");

        // 4. Check timing constraints
        uint256 blocksPassed = block.number - commitment.blockNumber;
        require(blocksPassed >= minRevealDelay, "Reveal: Too early");
        require(blocksPassed <= maxRevealDelay, "Reveal: Too late");

        // 5. Check voucher not claimed yet and user has no existing vesting schedule
        require(!claimedVouchers[voucherId], "Reveal: Voucher already claimed");
        require(vestingSchedules[user].totalAmount == 0, "Reveal: User already has vesting schedule
            ");

        // 6. Calculate leaf hash
        bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(voucherId, amount))));

        // 7. Verify Merkle proof
        require(MerkleProof.verify(merkleProof, merkleRoot, leaf), "Reveal: Invalid Merkle proof");

        // 8. Mark as revealed and claimed (both voucherId and leafHash)
        commitment.revealed = true;
        claimedVouchers[voucherId] = true;
        claimedLeafHashes[leaf] = true;

        // 9. Create vesting schedule and release tokens
        _createAndWithdrawImmediatelyReleasable(user, amount);

        emit Revealed(user, voucherId, amount);
    }
```

Listing 2.4: src/Q101AirdropVesting.sol

Impact Malicious actors can claim airdrops on behalf of users who provide valid airdrop information outside the valid delay range.

Suggestion Revise the logic accordingly.

Clarification from BlockSec If users call the function `reveal()` before `minRevealDelay`, the transaction will revert, exposing their `voucherId` on-chain. This creates a risk where attackers could exploit the leaked identifier to claim the user's airdrop. However, since `minRevealDelay` is an intentional safeguard enforced by the official frontend to prevent front-running attacks, this risk does not exist under normal user interactions.

2.2 Recommendation

2.2.1 Validate configured `startTime` in function `configureAirdrop()`

Status Confirmed

Introduced by `Version 1`

Description In contract `Q101AirdropVesting`, function `configureAirdrop()` sets core vesting logic, which cannot be modified once initialized. However, the function lacks sufficient validation for the variable `startTime` for all vesting schedules. Specifically, the input `_startTime` can be set as a past timestamp, allowing the vesting calculation for cliff and linear stages to begin from that historical point. Consequently, users who reveal may directly receive a significant or even the full vesting amount, violating the intended gradual release mechanism.

```
function configureAirdrop(
    uint64 _startTime,
    bytes32 _merkleRoot,
    uint256 _vestingDuration,
    uint256 _cliffDuration,
    uint256 _immediateReleaseRatio,
    uint256 _cliffReleaseRatio,
    VestingFrequency _vestingFrequency,
    uint256 _minWithdrawInterval,
    uint256 _minWithdrawAmount
) external onlyOwner {
    // ============ Validation ============

    // Can only be called once (when merkleRoot is not set)
    require(merkleRoot == bytes32(0), "Airdrop already configured");

    // Validate startTime
    require(_startTime > 0, "Invalid start time");

    // Merkle root must be non-zero
    require(_merkleRoot != bytes32(0), "Invalid merkle root");

    // Vesting parameters validation
    require(_vestingDuration > 0, "Invalid vesting duration");
    require(_minWithdrawInterval > 0, "Invalid min withdraw interval");
    require(_minWithdrawAmount > 0, "Invalid min withdraw amount");

    // Release ratios validation
    require(
```

```
        _immediateReleaseRatio + _cliffReleaseRatio <= RATIO_PRECISION,
        "Immediate + Cliff ratio must <= 100%"
    );
```

Listing 2.5: src/Q101AirdropVesting.sol

Suggestion Enforce that the configured `startTime` must be greater than or equal to the current block timestamp.

Feedback from the project The project confirmed that this is an intended business logic design. Allowing the configuration of past timestamps supports more airdrop scenarios, such as retrospective attribution.

2.2.2 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Q101AirdropVesting`, the code declares and assigns an immutable trusted forwarder variable, intending to support ERC2771. This logic is redundant because the parent contract `ERC2771ContextUpgradeable` already receives and manages the forwarder address upon construction.

```
/// @notice Trusted forwarder for Gelato Relay (ERC2771)
/// @custom:oz-upgrades-unsafe-allow state-variable-immutable state-variable-assignment
address private immutable _trustedForwarder;
```

Listing 2.6: src/Q101AirdropVesting.sol

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor(address trustedForwarder_) ERC2771ContextUpgradeable(trustedForwarder_) {
    _trustedForwarder = trustedForwarder_;
    _disableInitializers();
}
```

Listing 2.7: src/Q101AirdropVesting.sol

Suggestion Remove the redundant assignment.

2.2.3 Fix conflicts in documentation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `Q101AirdropVesting`, function `updateMerkleRoot()` allows modifying the Merkle root for an airdrop.

However, several places in the documentation state that the Merkle root is immutable, which conflicts with contract `Q101AirdropVesting`'s implementation.

```
function updateMerkleRoot(bytes32 _merkleRoot) external onlyOwner {
    require(merkleRoot != bytes32(0), "Must call configureAirdrop first");
    require(_merkleRoot != bytes32(0), "Invalid merkle root");
```

```

        bytes32 oldRoot = merkleRoot;
        merkleRoot = _merkleRoot;

        emit MerkleRootUpdated(oldRoot, _merkleRoot);
    }
```

Listing 2.8: src/Q101AirdropVesting.sol

```
#### Security Highlights:
- Merkle root can only be set once (immutable after initialization)
```

Listing 2.9: README.md

```
1. **One-Time Merkle Root**: Cannot update Merkle root after initial setup
   - Design: Prevents unauthorized changes
   - Workaround: Deploy new vesting contract for new distributions
```

Listing 2.10: README.md

Suggestion Fix the incorrect description in the documentation.

2.2.4 Optimize ownership grant logic

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `initialize()`, the ownership initialization and transfer on lines 49 and 55 are redundant, incurring unnecessary gas costs for storage writes. The ownership can be granted directly to address `gnosisSafe` via a single invocation to function `__Ownable_init()`.

```
49        __Ownable_init(msg.sender);
50        __Pausable_init();
51
52        _mint(gnosisSafe, TOTAL_SUPPLY);
53
54        // Transfer ownership to Gnosis Safe
55        transferOwnership(gnosisSafe);
```

Listing 2.11: src/Q101Token.sol

Suggestion Use `__Ownable_init(gnosisSafe)` to grant ownership.

2.3 Note

2.3.1 Ensure secure generation of vouchers

Introduced by `Version 1`

Description In the contract `Q101AirdropVesting`, a user's claim is verified against a Merkle root using `voucherId` as a unique identifier. Since `voucherId` is not bound to the user's address, the project should implement its stringent generation to ensure claim integrity and prevent unauthorized access. The voucherId must be generated offline in a manner that guarantees high entropy, confidential distribution, and global uniqueness, ensuring that it cannot be derived from public parameters or reused.

Feedback from the project The project acknowledged this note.

2.3.2 Potential centralization risks

Introduced by Version 1

Description In this project, privileged roles (e.g., owner) can conduct sensitive operations, which introduces potential centralization risks. For example, the owner controls critical airdrop parameter configurations (e.g., via functions configureAirdrop(), updateMerkleRoot(), updateRevealDelay(), and updateWithdrawRestrictions()) and can also invoke the function emergencyWithdraw() to extract all contract-held tokens. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The project confirmed that all sensitive operations, including configureAirdrop(), updateMerkleRoot(), updateRevealDelay(), and updateWithdrawRestrictions(), are managed via a multi-signature wallet, requiring authorization from multiple signatories for execution.

2.3.3 Proxy deployment and implementation binding should be atomic

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.

Feedback from the project The project acknowledged this note and will ensure the implementation is correctly bound upon deployment.

2.3.4 Ensure sufficient token balances in contract Q101AirdropVesting

Introduced by Version 1

Description In contract Q101AirdropVesting, there is no explicit logic to accept tokens for airdrop allocation. Therefore, the project should manually transfer tokens to the contract and maintain sufficient balances to fulfill all airdrop claims.

Feedback from the project The project acknowledged this note.

2.3.5 Security of Gelato integration

Introduced by `Version 1`

Description The contract `Q101AirdropVesting` integrates Gelato to allow users to make gas-less invocations (specifically for the functions `commit()`, `reveal()`, and `withdraw()`), with the gas sponsored by the project. To prevent malicious depletion of the sponsorship funding pool, the project should limit the user call frequency on the backend.

Feedback from the project The project acknowledged this note.

2.3.6 Merkle tree modification should only add new vouchers

Introduced by `Version 1`

Description In the contract `Q101AirdropVesting`, the function `updateMerkleRoot()` allows the project team to add new `voucherId` and `amount` pairs. To ensure that the update operation does not affect the eligibility of existing users, the new Merkle tree should include all previously valid leaf nodes when calculating the new `merkleRoot`.

Feedback from the project The project acknowledged this note.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS