QuillAudits

AUDIT REPORT

July 2025

For

BigWater

Table of Content

Executive Summary

Project Name	BigWater
Protocol Type	DePIN
Project URL	https://bigwater.io/
Overview	BigWater is a protocol consisting of a DeviceRegistry contract that allows users to register IoT devices and mint corresponding NFTs as proof of ownership, paired with a DePINStaking contract where users stake tokens into a reward pool that distributes fixed 10 BIGW rewards based on external performance scores from participants' registered devices.
Audit Scope	The scope of this Audit was to analyze the BigWater Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/BigWater-Protocol/BigWater-Smart-Contracts
Branch	main
Contracts in Scope	DePINStaking.sol, DeviceRegistry.sol, RewardDistribution.sol, BigWaterToken.sol, BigWaterNFT.sol
Commit Hash	6757311c85177bb359f58609efb8450657c96cf8
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	18th July 2025 - 24th July 2025
Updated Code Received	25th July 2025
Review 2	25th July 2025 - 26th July 2025
Fixed In	081af01661cef928d92f13e72e9ec99877970292

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity

5
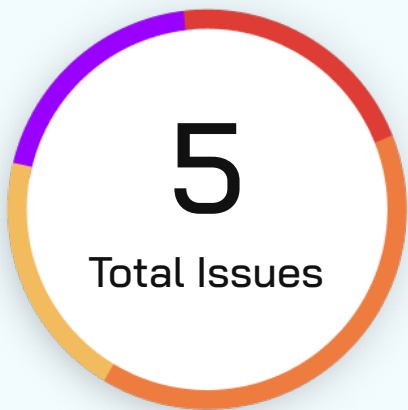
Total Issues

■ Critical	0	(0%)
■ High	1	(20.0%)
■ Medium	2	(40.0%)
■ Low	1	(20.0%)
■ Informational	1	(20.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	0	1	2	1	1

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	distributeRewards can be called by anyone	**High**	**Resolved**
2	Arbitrary TokenURI inputs will lead to JSON injection and phishing attacks	**Medium**	**Resolved**
3	Risk of DOS for the participants array	**Medium**	**Resolved**
4	Use Ownable2Step instead of Ownable	**Low**	**Resolved**
5	Risk of hash collision by using abi.encodePacked with dynamic data type	**Informational**	**Resolved**

Checked Vulnerabilities

✓ Access Management

✓ Arbitrary write to storage

✓ Centralization of control

✓ Ether theft

✓ Improper or missing events

✓ Logical issues and flaws

✓ Arithmetic Computations Correctness

✓ Race conditions/front running

✓ SWC Registry

✓ Re-entrancy

✓ Timestamp Dependence

✓ Gas Limit and Loops

✓ Exception Disorder

✓ Gasless Send

✓ Use of tx.origin

✓ Malicious libraries

✓ Compiler version not fixed

✓ Address hardcoded

✓ Divide before multiply

✓ Integer overflow/underflow

✓ ERC's conformance

✓ Dangerous strict equalities

✓ Tautology or contradiction

✓ Return values of low-level calls

- ✔ **Missing Zero Address Validation**
- ✔ **Private modifier**
- ✔ **Revert/require functions**
- ✔ **Multiple Sends**
- ✔ **Using suicide**
- ✔ **Using delegatecall**

- ✔ **Upgradeable safety**
- ✔ **Using throw**
- ✔ **Using inline assembly**
- ✔ **Style guide violation**
- ✔ **Unsafe type inference**
- ✔ **Implicit visibility level**

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

■ Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

■ High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

■ Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

■ Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

■ Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

	🟥 High	🟧 Medium	🟨 Low
🟥 **High**	Critical	High	Medium
🟧 **Medium**	High	Medium	Low
🟨 **Low**	Medium	Low	Low

(Likelihood)

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.

- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.

- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.

- Medium - only a conditionally incentivized attack vector, but still relatively likely.

- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

distributeRewards can be called by anyone   `Resolved`

Path

DePINStaking.sol

Function

`distributeRewards`

Description

The distributeRewards function is responsible for distributing reward emission to participants. The function is meant to be called by BigWater protocol. But since the function as no access control and is marked as external, this function is callable by anyone at the moment.

Impact

A malicious user can call this multiple times potentially disrupting the reward distribution behaviour.

Likelihood

Likelihood is high as this is callable by anyone.

Recommendation

Ensure that this is only callable by owner.

Medium Severity Issues

Arbitrary TokenURI inputs will lead to JSON injection and phishing attacks

Resolved

Path

DeviceRegistry.sol

Path

`registerDevice`

Description

Whenever a user registers device , the tokenURI can be an arbitrary input. Since there is no restriction on what this URI can be, it can lead to problems such as URI pointing to malicious content, JSON injection, phishing attacks by impersonating different NFTs, and gas griefing attacks with a large URI.

Impact

The impact of this depends upon input sanitization of the front-end system. If sanitization is not there, it can lead to cross site scripting, phishing etc.

Recommendation

Do not allow arbitrary data as tokenURI. Either make sure URI is system generated or a list of a mapping of valid URIs.

Risk of DOS for participants array   **Resolved**

Path

RewardDistribution.sol

Path

`submitScore`

Description

Whenever submitScore is called, if the owner is not a participant before, the function pushes that address to participants array.
However, given that there are enough user, the array could be so large that when distributeRewards is called, and loops through, the gas consumption could outweigh block gas limit causing denial of service.

Recommendation

Either limit the number of users that can be processed in an array at once and implement a function to remove participants if the array becomes large, or avoid using it.

Low Severity Issues

Use Ownable2Step instead of Ownable	Resolved

Description

The contracts inherit from Openzeppelin's Ownable library that allows instantaneous ownership transfers. This is associated with a risk of transferring ownership to an unintended address causing hijack of critical functionality or making them unusable if transferred to zero address.

Recommendation

Use Ownable2Step library instead of ownable.

Informational Issues

Risk of hash collision by using abi.encodePacked with dynamic data type

Resolved

Path

DeviceRegistry.sol

Function Name

registerDevice

Description

The registerDevice function uses abi.encodePacked on a string to create a device id hash. It is generally not recommended to use abi.encodePacked on dynamic data types as it can lead to hash collision.

Recommendation

Use **abi.encode** instead

Functional Tests

19

Some of the tests performed are mentioned below:

- ✔ Successfully register a device
- ✔ Consistent reward distribution calculation

Threat Model

Contract	Function	Threats
DeviceRegistry.sol	registerDevice	**Inputs** - owner: Control: Fully controlled by caller, Constraints: Must not be address(0), Impact: Address that will own the device and NFT - deviceId: Control: Fully controlled by caller, Constraints: Must not be empty string, Impact: Unique identifier for the device - tokenURI: Control: Fully controlled by caller, Constraints: Must not be empty string, Impact: Metadata URI for the NFT Control: No access control - anyone can register devices for any owner Constraints: Basic validation for zero address and empty strings Impact: Creates device registry entry and mints NFT to specified owner

Contract	Function	Threats
		Branches and Code Coverage Intended Branches - Should validate owner is not zero address - Should validate deviceId is not empty - Should validate tokenURI is not empty - Should check device is not already registered using hash - Should mint NFT successfully and receive valid token ID - Should store device data in mapping - Should add deviceId to owner's device list - Should track new owners in registeredOwners array - Should emit DeviceRegistered event Negative Behavior - Should not allow registration with zero address owner - Should not allow registration with empty deviceId - Should not allow registration with empty tokenURI - Should not allow duplicate device registration - Should not proceed if NFT minting fails

Contract	Function	Threats
	batchRegisterDevices	**Security Concerns** - Critical Access Control Flaw: Anyone can register devices for any address without permission - Hash Collision Risk: Uses keccak256(abi.encodePacked(deviceId)) which is vulnerable to hash collisions - URI Injection: tokenURI is not validated, allowing potential metadata manipulation Allows batch registration of multiple devices with the same security flaws as single registration. **Inputs** - owners: Control: Fully controlled by caller, Constraints: Array length must match other arrays, Impact: Addresses that will own devices - deviceIds: Control: Fully controlled by caller, Constraints: Array length must match, Impact: Device identifiers - tokenURIs: Control: Fully controlled by caller, Constraints: Array length must match, Impact: NFT metadata URIs

Contract	Function	Threats
		Control: No access control - anyone can batch register for any owners Constraints: Array length validation only Impact: Mass device registration with potential for large-scale attacks

Branches and Code Coverage

Intended Branches

- Should validate all arrays have equal length
- Should successfully register all devices in sequence
- Should handle partial failures gracefully

Negative Behavior

- Should not allow mismatched array lengths
- Should not continue if any registration fails

Security Concerns

- Amplified Access Control Issues: Same flaws as registerDevice but at scale
- Gas Griefing: Large arrays can cause excessive gas consumption
- Partial State Corruption: If one registration fails, previous ones remain, causing inconsistent state |

Contract	Function	Threats
DePINStaking.sol	stake	Allows users to stake tokens into the reward pool. **Inputs** - amount: Control: Fully controlled by caller, - Constraints: Must be > 0, Impact: Number of tokens to stake Control: Public function, anyone can stake Constraints: Amount must be positive, user must have sufficient balance and allowance Impact: Increases totalStaked and transfers tokens to contract **Branches and Code Coverage** Intended Branches - Should validate amount is greater than zero - Should successfully transfer tokens from user to contract - Should update totalStaked correctly - Should emit Staked event Negative Behavior - Should not allow zero amount staking - Should not proceed if token transfer fails - Should not allow staking without sufficient balance/allowance

Contract	Function	Threats
	distributeRewards	**Security Concerns** - No Staking Records: Contract doesn't track individual stakes, only total - No Withdrawal Mechanism: Users cannot withdraw their staked tokens - Permanent Lock: Staked tokens appear to be locked forever Distributes fixed 10 BIGW rewards to participants based on external scores. - Control: Public function, anyone can trigger distribution - Constraints: Must have participants, total score > 0, sufficient pool balance - Impact: Distributes FIXED_EMISSION (10 ether) to participants and reduces totalStaked

Contract	Function	Threats
		Branches and Code Coverage Intended Branches - Should fetch participants from external contract - Should calculate total score across all participants - Should validate sufficient pool balance - Should distribute rewards proportionally to scores - Should reduce totalStaked by FIXED_EMISSION - Should emit YieldDistributed events Negative Behavior - Should not distribute if no participants exist - Should not distribute if total score is zero - Should not distribute if insufficient pool balance - Should not transfer rewards if token transfers fail

Contract	Function	Threats
		Security Concerns - Critical Logic Error: totalStaked is reduced by - FIXED_EMISSION regardless of actual rewards distributed (participants with zero scores get no rewards but total is still reduced) - External Dependency Risk: Relies entirely on external rdc contract for participant list and scores - Oracle Manipulation: External reward contract could be manipulated to affect distributions - Griefing Attack: Anyone can trigger distribution, potentially at inappropriate times - Rounding Errors: Integer division may cause reward dust to accumulate - No Reentrancy Protection: External calls to rdc could potentially cause reentrancy issues

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of BigWater. We performed our audit according to the procedure described above.

issues of High, Medium, Low and Informational severity were found. BigWater team resolved all the issues mentioned

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

July 2025

For

BigWater

QuillAudits

Canada, India, Singapore, UAE, UK