



Security Audit

Report for Reward Distribute Platform

Date: July 20, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction	1
1.1 About the Audit Target	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 Recommendation	4
2.1.1 Align the implementation with the declared compiler version	4
2.2 Note	4
2.2.1 Potential centralization risks	4

Report Manifest

Item	Description
Client	bitgetwallet
Target	Reward Distribute Platform

Version History

Version	Date	Description
1.0	July 20, 2026	First release

Signature

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 Reward Distribute Platform of bitgetwallet.

Reward Distribute Platform is a Solidity-based campaign reward vault. It lets creators create reward campaigns, sponsors deposit ERC20 budgets, publishers release per-epoch Merkle roots, and users claim cumulative rewards with Merkle proofs. The contract tracks committed, claimed, and locked funds per campaign/token, supports cancellation and sponsor refunds, and includes owner/operator governance for role updates, pausing, ERC20/ETH rescue, and emergency campaign sponsor-fund recovery. The design relies on off-chain Merkle generation and privileged roles, especially the publisher and operator, for correct reward distribution and incident handling.

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 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
reward-distribute-platform	Version 1	a9f2393c26e057a5b06c9eec61439604f2396cad

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.

¹<https://github.com/bitgetwallet/reward-distribute-platform>

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	High	High	Medium
	Low	Medium	Low
		High	Low
		Likelihood	

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.

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

³<https://cwe.mitre.org/>

Chapter 2 Findings

In total, we have **one** recommendation and **one** note.

- Recommendation: 1
- Note: 1

ID	Severity	Description	Category	Status
1	-	Align the implementation with the declared compiler version	Recommendation	Confirmed
2	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Recommendation

2.1.1 Align the implementation with the declared compiler version

Status Confirmed

Introduced by [Version 1](#)

Description The contract declares `pragma solidity 0.8.20`, indicating compatibility with Solidity 0.8.20 and later 0.8.x compiler versions. However, the implementation of function `safeTransferETH()` in contract `CampaignVaultBase` uses `require(success, ETHTransferFailed())`, which relies on a compiler feature introduced after Solidity 0.8.20. As a result, the contract cannot be compiled with some compiler versions permitted by the declared pragma, such as Solidity 0.8.20. Although this does not affect runtime behavior when compiled with a sufficiently recent compiler, aligning the implementation with the declared compiler version improves build consistency and avoids confusion for integrators and developers.

```
98  function safeTransferETH(address to, uint256 value) internal {
99      (bool success,) = to.call{value: value}(new bytes(0));
100      require(success, ETHTransferFailed());
101  }
```

Listing 2.1: src/CampaignVaultBase.sol

Suggestion Use an explicit custom-error revert that is compatible with Solidity 0.8.20.

Feedback from the project The project has decided not to make any changes. To support multi-chain deployment, the contracts will be compiled and deployed using the appropriate compiler versions for each target chain.

2.2 Note

2.2.1 Potential centralization risks

Introduced by [Version 1](#)

Description In this project, several privileged roles, including the `owner`, `operator`, `publisher`, and campaign `creator`, can perform sensitive operations, which introduces potential centralization risks. For example, the `owner` can update the `operator`, `safe`, and `publisher` addresses and cancel campaigns; the `operator` can pause or unpause the protocol and trigger emergency rescue/refund functions; the `publisher` can publish Merkle roots that determine reward distribution; and the campaign creator can cancel campaigns and initiate refunds of unclaimed budgets to the sponsor. If the private keys of these privileged accounts are lost, compromised, or maliciously used, it could affect campaign execution, reward distribution, user claimability, and fund recovery, posing operational and trust risks to the protocol.

