RISK AUDIT

for

PB&J
CONSULTING

on

Jun 02, 2025

FIDESIUM

Executive Summary

Report

7
/100

TOTAL
Low risk
June 09, 2025

24
/100

TOTAL
Low risk
June 02, 2025

Abstract

Fidesium's automated risk assessment service was requested to perform a risk posture audit on TriviTournament **contracts**

Repository Link: https://github.com/PBJ-JWeb3/Trivi-Contracts

Initial Commit Hash:

`896ffabc8fd1b715d599cc5ccf1f3d9640f0256e`

Issue Summary

Critical	High	Medium	Low	Info
0 Issues	~~2~~ 0 Issues	~~3~~ 1 Issues	~~1~~ 1 Issues	~~2~~ 1 Issues

Caveats

PBJ's codebase is generally well written, but does incur a handful of flaws.

Test Approach

Fidesium performed both Whitebox and Blackbox testing, as per the scope of the engagement, and relied on automated security testing.

Methodology

The assessment methodology covered a range of phases and employed various tools, including but not limited to the following:

- Mapping Content and Functionality of API
- Application Logic Flaws
- Access Handling
- Authentication/Authorization Flaws
- Brute Force Attempt
- Input Handling
- Source Code Review
- Fuzzing of all input parameter
- Dependency Analysis

Severity Definitions

Critical	The issue can cause large economic losses, large-scale data disorder or loss of control of authority management.
High	The issue puts users' sensitive information at risk or is likely to lead to catastrophic financial implications.
Medium	The issue puts a subset of users' sensitive information at risk, reputation damage or moderate financial impact.
Low	The risk is relatively small and could not be exploited on a recurring basis, or is low-impact to the client's business.
Informational	The issue does not pose an immediate risk but is relevant to security best practices or defence in Depth.

Risk Issues

Vulnerability	Description	Risk	Probability	Status
Data Corruption: Storage Slot Collision	The `TriviTournament` contract nests a mapping in a struct, which can lead to storage slot collision.	High	Medium	Resolved
DoS: Unbounded Loop	The `TriviTournament.cancelTournament` function iterates without a gas limit, and can be used to DOS the contract.	High	Medium	Resolved
One step ownership transfer	The `TriviTournament` contract relies on `Ownable` to manage ownership, which is not secure.	Medium	Medium	Active
Centralization	The `backendService` has significant modification rights over the contracts and their state.	Medium	Medium	Acknowledged
Missing bounds validation	The `enterTournament` does not validate against `maxPlayers`.	Medium	Medium	Resolved
State inconsistency: Partial Refund with cleanup	The `TriviTournament` contract does not verify refund completion before cleanup.	Low	Low	Active
Gas Vulnerability: Permanent Storage Bloat	The `TriviTournament` contract uses a mapping to store the tournaments.	Low	Low	Resolved
Gas Inefficiency: Repeated storage reads	The `TriviTournament` contract reads the `tournament` variable repeatedly.	Info	Info	Resolved
Gas Inefficiency: String Comparison as Existence Check	The `TriviTournament` contract uses a string comparison to check for existence.	Info	Info	Resolved
Gas Inefficiency: High write frequency storage	The `TriviTournament.Tournament` struct has fields with high write frequency.	Info	Info	Active

Risk Overview

Team Risk

Low risk: 1

No issues found in founding team

Doxxing Status	Team Experience	Risk Summary
Public	Highly relevant	Low

Smart Contract Risks

Risk summary: ~~27~~ 11

The contracts are well written, and secure with only a few minor issues..

Vulnerabilities Critical

Current scan criticals Clear

During this scan no critical security vulnerabilities were identified. The assessment covered all key components of the project, including smart contract logic, access controls, and potential attack vectors. While no critical issues were found, we recommend ongoing security monitoring and best practices to maintain the integrity and resilience of the system.

Vulnerabilities High

Data Corruption: Storage Slot Collision

Vulnerability severity: **High**

Vulnerability probability: **Medium**

The `TriviTournament` contract nests a mapping in a struct, which can lead to storage slot collision.

Nested Mappings and dynamic arrays in a struct do not use the struct's slot, instead they calculate the slot based on the hash of the struct and the mapping/array's key. If an attacker crafts a second tournament id that collides with the first tournament id, the second tournament will overwrite the first tournament's data. This can lead to manipulation, DoS, and, in extreme cases, protocol failure

Recommendations:

Separate the mapping and array from the struct, and use a different slot for the mapping.

```
mapping(string => mapping(address => bool)) public tournamentParticipants;
mapping(string => address[]) public tournamentPlayers;
```

Action Taken:

Resolved at commit: `896ffabc8fd1b715d599cc5ccf1f3d9640f0256e` by removing the nested mapping and array from the struct.

Vulnerabilities High

DoS: Unbounded Loop

Vulnerability severity: **High**

Vulnerability probability: **Medium**

The `TriviTournament.cancelTournament` function iterates without a gas limit, and can be used to DOS the contract. The function iterates over the `tournaments` array, and for each tournament, it iterates over the `players` array. If the `players` array is large, the function will run out of gas and revert. This can be used to DOS the contract, and prevent users from cancelling tournaments.

Recommendations:

- Add a gas limit to the function.
- Implement a pull over push strategy for the `players` array.

```
mapping(string => mapping(address => uint256)) public refunds;
...
function cancelTournament(string memory tournamentId) external {
    ...
    for (uint256 i = 0; i < tournament.players.length; i++) {
        refunds[tournamentId][tournament.players[i]] = tournament.entryFee;
    }
    tournament.isActive = false;
}

function claimRefund(string memory tournamentId) external nonReentrant {
    uint256 refundAmount = refunds[tournamentId][msg.sender];
    require(refundAmount > 0, "No refund available");
    refunds[tournamentId][msg.sender] = 0;
    triviToken.safeTransfer(msg.sender, refundAmount);
}
```

Action Taken:

Resolved at commit: `896ffabc8fd1b715d599cc5ccf1f3d9640f0256e` by batching the refunds and allowing backend to disperse refunds in batches to prevent DoS.

Vulnerabilities Medium

One Step Ownership Transfer

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

The `TriviTournament` contract relies on `Ownable` to manage ownership, which is not secure.

The `Ownable` pattern is vulnerable to a one step ownership transfer. This exposes these contracts to accidental ownership transfer to malicious or invalid wallets.

Recommendations:

Implement `Ownable2Step` to drive a two step ownership transfer. This will require applying `Upgradeable` independently.

Centralization

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

The `backendService` has significant modification rights over the contracts and their state.

Recommendations:

Ensure that these roles are tied to well maintained Multisig wallets, and consider implementing a timelock.

Action Taken:

Acknowledged by team who will ensure all multisig wallets are well maintained.

Missing bounds validation

Vulnerability severity: **Medium**

Vulnerability probability: **Medium**

The `enterTournament` does not validate against `maxPlayers`.

Recommendations:

Validate the `maxPlayers` parameter.

Action Taken:

Resolved at commit: `896ffabc8fd1b715d599cc5ccf1f3d9640f0256e` by adding a check for `maxPlayers`.

Vulnerabilities Low

State inconsistency: Partial Refund with cleanup

Vulnerability severity: **Low**

Vulnerability probability: **Low**

The `TriviTournament` contract does not verify refund completion before cleanup.

The partial batched refunds can lead to state inconsistency if the backend does not complete the refunds.

Recommendations:

Implement a check for refund completion before cleanup.

```
                            mapping(string => uint256) public totalRefundsDispersed;
                            mapping(string => uint256) public totalRefundsRequired;

                            function cancelTournament(string memory tournamentId)
        external
        onlyBackendService
        tournamentExistsCheck(tournamentId)
        nonReentrant
    {
        Tournament storage tournament = tournaments[tournamentId];

        require(tournament.isActive, "Tournament is not active");
        require(!tournament.isCompleted, "Tournament is already completed");

        tournament.isActive = false;
        tournamentCancelled[tournamentId] = true;

        totalRefundsRequired[tournamentId] = tournament.playerCount;
        totalRefundsDispersed[tournamentId] = 0;

        emit TournamentCancelled(tournamentId);
    }
    ....
    ....
    if (tournamentCancelled[tournamentId]) {
        require(
            totalRefundsDispersed[tournamentId] == totalRefundsRequired[tournamentId],
            "All refunds must be dispersed before cleanup"
        );
    }
```

Action Taken:

Vulnerabilities Low

Gas Vulnerability: Permanent Storage Bloat

Vulnerability severity: **Low**

Vulnerability probability: **Low**

The `TriviTournament` contract uses a mapping to store the tournaments.

This can lead to permanent storage bloat, and can be used to DOS or grief the contract via storage exhaustion in extreme cases.

Recommendations:

- Implement tournament cleanup
- Use incremental tournament ids
- For large player counts, use merkle trees.

Action Taken:

Resolved at commit: `896ffabc8fd1b715d599cc5ccf1f3d9640f0256e`.

Vulnerabilities Info

Gas Inefficiency: Repeated storage reads

Vulnerability severity: **Info**

Vulnerability probability: **Info**

The `TriviTournament` contract reads the `tournament` variable repeatedly.

Recommendations:

Cache the `tournament` reference.

```
Tournament storage tournament = tournaments[tournamentId];
```

Action Taken:

Resolved at commit: `896ffabc8fd1b715d599cc5ccf1f3d9640f0256e` by caching the `tournament` reference.

Gas Inefficiency: String Comparison as Existence Check

Vulnerability severity: **Info**

Vulnerability probability: **Info**

The `TriviTournament` contract uses a string comparison to check for existence.

`bytes(tournamentId).length > 0` is gas intensive.

Recommendations:

Use a separate existence mapping.

Action Taken:

Resolved at commit: `896ffabc8fd1b715d599cc5ccf1f3d9640f0256e` by using a separate existence mapping.

Gas Inefficiency: High write frequency storage

Vulnerability severity: **Info**

Vulnerability probability: **Info**

The `TriviTournament.Tournament` struct has fields with high write frequency.

These fields are written to frequently, and can lead to high gas costs. `totalPrizePool` and `playerCount`.

Recommendations:

Remove these fields from the struct. The gas savings on writes far outweigh the gas savings on computation/reads. You could also cache them on tournament completion, e.g.:

```
mapping(string => uint256) public completedTournamentPrize;
```

Ensure that all fields in the struct are packed correctly to make use of slot sizes (32 bytes)

Disclaimer

Disclaimer

This report is governed by the Fidesium terms and conditions.

This report does not constitute an endorsement or disapproval of any project or team, nor does it reflect the economic value or potential of any related product or asset. It is not investment advice and should not be used as the basis for investment decisions. Instead, this report provides an assessment intended to improve code quality and mitigate risks inherent in cryptographic tokens and blockchain technology.

Fidesium does not guarantee the absence of bugs or vulnerabilities in the technology assessed, nor does it comment on the business practices, models, or regulatory compliance of its creators. All services, reports, and materials are provided "as is" and "as available," without warranties of any kind, including but not limited to merchantability, fitness for a particular purpose, or non-infringement.

Cryptographic assets and blockchain technologies are novel and carry inherent technical risks, uncertainties, and the possibility of unpredictable outcomes. Assessment results may contain inaccuracies or depend on third-party systems, and reliance on them is solely at the Customer’s risk.

Fidesium assumes no liability for content inaccuracies, personal injuries, property damages, or losses related to the use of its services, reports, or materials. Third-party components are provided "as is," and any warranties are strictly between the Customer and the third-party provider.

These services and materials are intended solely for the Customer's use and benefit. No third party or their representatives may claim rights to or rely on these services, reports, or materials under any circumstances.