Color | RGBClash

Smart Contract Security Assessment

VERSION 0.9

AUDIT DATES: Feb 3-5, 2026

AUDITED BY: vard

Contents

ᛉ Introduction

Algiz conducted an independent review of the smart-contract system. This engagement focused on evaluating the correctness of core protocol logic, identifying vulnerabilities that could lead to loss of funds or protocol manipulation, and assessing resilience under adversarial conditions.

This report summarizes the issues identified during the review, provides severity classifications, and offers actionable recommendations aimed at strengthening the protocol's security posture. The findings presented here reflect the codebase at the specified commit and may not apply to future revisions.

ᛟ About Algiz

Algiz is an independent smart contract security team specializing in helping early-stage protocols launch with confidence. Our team has identified 30+ Critical, High and Medium severity vulnerabilities across DeFi protocols, GameFi projects, and cross-chain systems through competitive audits and security research.

We combine engineering depth with product thinking - reviewing protocols not only for correctness, but for operational risk, upgrade safety, and long-term sustainability. Our founder-to-founder approach means we're a security partner, not just a vendor.

ᚾ Disclaimer

This report documents Algiz's observations based on a time-boxed review of the supplied codebase at the specified commit. The presented conclusions reflect only this snapshot of the system.

While every reasonable effort has been made to identify vulnerabilities, **no security review can guarantee** the complete absence of bugs, exploits, or unexpected behaviors. Smart-contract systems operate in adversarial, permissionless environments, and security must be treated as an ongoing process.

The mitigation recommendations included in this report are provided as guidance based on the information available during the engagement. Because this was a time-boxed review, complex or large-scale remediation efforts may require extended follow-up. In cases where fixes introduce substantial refactoring or modify core logic, we strongly advise conducting an additional full review to ensure the changes do not introduce new risks.

Subsequent security reviews, bug-bounty programs, and continuous on-chain monitoring are strongly recommended to maintain long-term protocol safety.

Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Informational

Impact	Description
High	Leads to a significant material loss of assets in the protocol, significantly harms a group of users, or disrupts a core functionality.
Medium	Leads to a moderate material loss of assets in the protocol, moderately harms a group of users, or affects ancillary functionalities.
Low	Leads to a minor material loss of assets in the protocol or to any unexpected behavior with some of the protocol's functionalities, but does not meet the criteria for higher severity.

Likelihood	Description
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. Vectors that require larger amount of capital to exercise relative to the amount gained or disruption of the protocol.
Low	Has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

ᛉ Executive Summary

Algiz conducted a focused security assessment of Color's RGBClash smart contracts during development stage. The review covered the core gaming infrastructure: NFT minting and staking, multiplayer game lobbies, and tournament management. The assessment identified 3 Medium-severity issues and 8 Informational observations, with no Critical or High impact vulnerabilities discovered.

The Medium findings center on fund-handling safety across the protocol's two competitive modes:

[M-1] Hard-reverting ETH transfers inside loops create a denial-of-service vector: a single non-receivable participant can brick tournament cancellation refunds and reward distribution.

[M-2] Flexible-start tournaments (`startTime == 0`) leave players with no self-service refund path once the registration deadline passes, making fund recovery entirely dependent on owner action.

[M-3] Replayable game rooms retain their original `creationTime` , allowing `withdrawOldRooms()` to sweep entry fees from rooms with recently joined players and active sessions.

A common theme across these findings is the reliance on owner intervention as the fallback for edge-case fund recovery, rather than providing players with autonomous safety mechanisms.

Informational items address missing input validation on tournament creation, unvalidated winner addresses, unfunded NFT reward recording, an incomplete boost mechanism, inconsistent event emissions, and gas optimization opportunities through custom errors.

The Color team was responsive throughout the review process. Of the 11 issues identified, 9 were resolved and 2 were acknowledged. M-3 was acknowledged with the understanding that lobbies are not intended for multi-day use; I-5 (incomplete boost mechanism) was acknowledged as the feature is not yet fully developed.

About Protocol

RGBClash is a blockchain-based multiplayer PvP gaming platform where players mint NFTs, stake them for rewards, and compete in games and tournaments with ETH entry fees. The system uses off-chain game servers with on-chain settlement for prize distributions. This is an initial quotation for a security audit of the contracts defined in the scope below.

Scope and Methodology

The engagement involved a review of the following targets:

Repository	https://github.com/color-xyz/contracts/
Commit Hash	5759d4b80fa0473c65bd345357a28290ec835472
Files	contracts/*.sol

Target	nSLOC
contracts/ColorNft.sol	192
contracts/ColorNftPouchEth.sol	103
contracts/MultiGameLobbyEth.sol	194
contracts/TournamentManager.sol	270
Total	**759**

Issues Found

Severity	Total	Acknowledged	Resolved
Critical Risk	-	-	-
High Risk	-	-	-
Medium Risk	3	1	2
Low Risk	-	-	-
Informational	8	1	7
Total Issues	**11**	**2**	**9**

Findings

ID	Title	Severity	Status
M-1	Push ETH transfers with hard reverts can brick tournament cancellation and reward distribution	Medium	Resolved
M-2	Flexible-start tournaments (startTime == 0) can leave players with no self-service refund after registration closes	Medium	Resolved
M-3	withdrawOldRooms() can reclaim entry fees from active replayed rooms by using stale creationTime	Medium	Acknowledged
I-1	Missing parameters validation on tournament creation	Informational	Resolved
I-2	TournamentManager does not validate winners[] are registered players	Informational	Resolved
I-3	MultiGameLobbyEth can record NFT rewards without funding the ColorNftPouch	Informational	Resolved
I-4	withdraw() is marked payable unnecessarily	Informational	Resolved
I-5	boost mechanism appears unfinished / unused	Informational	Acknowledged
I-6	Consider replacing revert strings with custom errors	Informational	Resolved
I-7	Missing events for privileged state changes and fund withdrawals	Informational	Resolved
I-8	Wrong event emitted in ColorNft.sol:approve()	Informational	Resolved

ᚦ Critical

No critical vulnerabilities were identified during this assessment.

ᚺ High

No high severity vulnerabilities were identified during this assessment.

ᛗ Medium

The following Medium severity vulnerabilities were identified during this assessment:

[M-1] Push ETH transfers with hard reverts can brick tournament cancellation and reward distribution

IMPACT:	Medium	**SEVERITY:**	Medium
LIKELIHOOD:	Medium	**STATUS:**	Resolved

Target

`TournamentManager.sol:cancelTournament()` ; `TournamentManager.sol: _distributeRewards() / distributeFinalRewards()`

Description

`TournamentManager` sends ETH via `.call{value: ...}("")` inside loops and enforces `require(success, ...)`. If any recipient is a contract that cannot receive ETH (reverting `receive()` / `fallback()`), the entire function reverts.

This creates a DoS vector:

- `cancelTournament()`: one non-receivable registered player blocks the whole cancellation flow (mass refunds + incentive refund).
- `distributeFinalRewards()` → `_distributeRewards()`: one non-receivable winner blocks finalization and payouts in that call.

Because these are core lifecycle paths, a single malicious participant (or contract) can prevent intended refunds/finalization and force the owner into “workarounds” that change semantics and can strand funds.

Root cause

The contract enforces ETH transfers with a hard revert inside recipient loops:

```
(bool success, ) = players[i].call{value: tournament.entryFee}("");
require(success, "Refund failed");
```

Because this pattern is executed in a loop, one failing recipient reverts the entire operation, rolling back all state changes and preventing partial progress.

Impact

Cancellation path can be permanently blocked, preventing the intended "cancel and refund all" mechanism from succeeding.

Finalization path can be blocked if a non-receivable winner is included (either accidentally, or maliciously if a contract address is chosen as winner).

The owner can attempt a workaround by using `distributeFinalRewards()` to pay out "refund-like" amounts only to receivable players (and omit the malicious address), but this:

- Changes semantics: the tournament becomes "finalized" rather than "cancelled", deviating from the intended lifecycle.
- Incentive handling becomes owner-mediated
- Omitted/non-receivable entrant requires manual reallocation

Recommendation

Adopt pull payments (best practice):

- On cancellation/finalization, do not push ETH to all recipients in a loop.
- Record each recipient's claimable amount and provide `claimRefund(tournamentId)` / `claimReward(tournamentId)` for withdrawals.
- Alternatively, implement "credit-on-failure": if a transfer fails, do not revert; store the amount as claimable and emit an event.

[M-2] Flexible-start tournaments (startTime == 0) can leave players with no self-service refund after registration closes

IMPACT:	High	**SEVERITY:**	Medium
LIKELIHOOD:	Low	**STATUS:**	Resolved

Target

`TournamentManager.sol`

Description

For tournaments created in flexible-start mode (`startTime == 0`), once `registrationDeadline` passes, players cannot exit via `unregisterPlayer()` and also cannot access the abandoned refund

mechanism because `claimAbandonedTournamentRefund()` requires `startTime != 0`. If the owner never starts/cancels/finalizes, players have no self-service path to recover entry fees.

Root cause

The contract explicitly supports `startTime == 0` at creation:

```
require(registrationDeadline < startTime || startTime == 0, "Registration must end before start");
```

but the two player exit paths exclude that mode after the deadline:

- `unregisterPlayer()` requires:

```
require(block.timestamp < tournament.registrationDeadline, "Registration deadline passed");
```

- `claimAbandonedTournametRefund()` requires:

```
require(tournament.startTime != 0, "No start time set");
require(block.timestamp >= tournament.startTime + 8 hours, "8 hours not passed since start time");
```

There is no function to later set `startTime` for a flexible-start tournament, meaning it can never transition into a state where "abandoned refund" becomes available.

Impact

Players' entry fees can be locked indefinitely in flexible-start tournaments after registration closes, making recovery dependent on owner action (start/cancel/finalize). This undermines the intended "abandoned tournament refund" safety valve for `startTime == 0` mode

Recommendation

Add a refund fallback for flexible-start tournaments:

- If `startTime != 0` - existing rule (`startTime + 8 hours`)
- If `startTime == 0` - allow after (`registrationDeadline + X hours / days`)

[M-3] withdrawOldRooms() can reclaim entry fees from active replayed rooms by using stale creationTime

IMPACT:	Medium	**SEVERITY:**	**Medium**
LIKELIHOOD:	Medium	**STATUS:**	**Acknowledged**

Target

`MultiGameLobbyEth.sol`

Description

`MultiGameLobbyEth` rooms are replayable: after `_endGame()`, the room resets `(gameStartTime = 0`, `gameId++`, players cleared) and can accept new players again. However, `room.creationTime` is set only once at room creation and never updated for new sessions.

`withdrawOldRooms()` determines "abandoned" rooms using only `room.creationTime` against `cutoff = now - 15 days`. If a room was created long ago but is currently hosting a fresh session (players recently joined, game possibly in progress), it is still treated as "old" and can be swept.

When swept, the function computes `roomValue = entryPrice * players.length`, clears players, resets game state, and transfers the value to the owner — even if those players joined recently.

Root cause

Mismatch between replayable-room design and abandonment criterion:

- Replayability defines "current activity" by per-session state (`players`, `gameStartTime`, `gameId`).
- Reclamation defines "abandoned" by immutable room age (`creationTime`) instead of last activity / current session time, and does not gate on `gameStartTime`.

Impact

- Recent joiners in an old room can lose their full entry fees if the owner runs `withdrawOldRooms()`.
- Active sessions can be disrupted (players wiped; game state reset).
- because `withdrawOldRooms()` only scans forward from `lastWithdrawnRoomId` and never revisits earlier `roomIds`, rooms that become abandoned after they've been passed may never be reclaimed by this mechanism.
- This is an admin footgun.

Recommendation

Use activity-based abandonment, not room creation time (e.g., add `lastActivityTime` and compare that to cutoff).

Low

No low severity vulnerabilities were identified during this assessment.

Informational

[I-1] Missing parameters validation on tournament creation

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Resolved

Target
`TournamentManager.sol:createTournament()`

Description
`createTournament()` is `onlyOwner`, but it lacks basic sanity checks on critical parameters (deadlines, start time, player caps, and percentage splits). Misconfiguration can create tournaments that are immediately invalid, impossible to operate, or that put users into unexpected states (e.g., no ability to unregister, immediate "started" semantics, or payout math that reverts).

Recommendation
Implement explicit checks in `createTournament()` and document the intended parameter constraints.

[I-2] TournamentManager does not validate winners[] are registered players

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Resolved

Target
`TournamentManager.sol`

Description
`TournamentManager` distributes ETH rewards to addresses provided in `winners[]` without verifying they are registered players / room participants. This is inconsistent with `MultiGameLobbyEth`, which explicitly restricts direct winnings recipients to room players (or the `ColorNftPouch`).

Impact
Assuming `Owner` is trusted, this is not an external exploit path. However, it is an operational footgun that can:

- allow accidental payouts to wrong / non-participant addresses due to UI, offchain script, or input mistakes;

- create confusion because different modules enforce different payout invariants.

  Recommendation
  Implement validation for `winners` are registered players in the tournament

[I-3] MultiGameLobbyEth can record NFT rewards without funding the ColorNftPouch

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Resolved

Target
`MultiGameLobbyEth.sol`

Description
`MultiGameLobbyEth` distributes ETH to `recipients[]` and separately records NFT reward allocations in `ColorNftPouch` via `distributeRewards(ids, idAmounts)`. However, unlike `TournamentManager`, it does not explicitly transfer ETH to the pouch for those NFT allocations, nor does it validate that the pouch was included in `recipients[]` with an amount matching `sum(idAmounts)`. This allows unfunded NFT rewards to be recorded on-chain due to an operational / input mistake.

Impact
This is not an external exploit path under a trusted-owner assumption. However, it is a meaningful operational footgun and invariant inconsistency.

If rewards are recorded but the pouch is underfunded, `claimRewards()` may revert and `usntakeNft()` will be blocked due to `rewards == 0` requirement, temporarily locking the NFT until the pouch is funded.

Recommendation
Consider implementing similar solution as in `TournamentManager` for ensuring there is enough ETH for rewards within the tx.

[I-4] withdraw() is marked payable unnecessarily

IMPACT:	Low	**SEVERITY:**	Informational

LIKELIHOOD:	Low	**STATUS:**	Resolved

Target
`ColorNft.sol`

Description
`withdraw()` dis marked `payable` even though it only forwards `address(this).balanc` to the owner and does not rely on `msg.value`, not receiving funds.

Impact
No impact. Confusing and unnecessary.

Recommendation
Remove `payable` modifier for `withdraw()`

[I-5] boost mechanism appears unfinished / unused

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Acknowledged

Target
`ColorNftPouchEth.sol`

Description
`ColorNftPouchEth` defines boost-related state (`NFT.boost`, `totalBoost`) and events (`DepositBoost`, `BoostsWithdrawn`), but the boost mechanism is not clearly used/enforced in reward accounting (or is only partially wired). This suggests an incomplete feature.

Impact
No direct impact, but unfinished/unused reward-weighting variables cannot be assessed at this moment.

Recommendation
Finish the boost-mechanism and perform another review.

[I-6] Consider replacing revert strings with custom errors

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Resolved

Target

`ColorNft.sol` , `ColorNftPouchEth.sol` , `MultiGameLobbyEth.sol` , `TournamentManager.sol`

Description

Across the codebase, input validation and access checks rely on `require(.., "revert string")` rather than custom errors. A review of the contracts identified 104 `require` statements with 77 unique revert strings, and 2,100 total characters of revert text, with 0 custom errors defined.

Impact

Non-security issue but impacts deployment size, revert-path gas usage, and consistency.

Recommendation

Adopt custom errors as the default pattern for revert reasons and standardize a shared set of reusable errors across contracts. Prefer:

- `if (!con) revert CustomError(..)` for complex/multiple conditions,

or

- `require(cond, CustomError(..)` for simple checks

[I-7] Missing events for privileged state changes and fund withdrawals

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Resolved

Target

`ColorNft.sol` , `ColorNftPouchEth.sol` , `MultiGameLobbyEth.sol` , `TournamentManager.sol`

Description

Several privileged functions and critical accounting actions across the contracts do not emit events.

Impact
While this does not introduce exploit path, it reduces observability for operators, indexers, and offchain monitoring.

Recommendation
Emit events in:

- ColorNft: `withdraw()`, `setPrice()`
- ColorNftPouchEth: `setAuthorizedDistributor()`
- MultiGameLobbyEth: `withdrawOldRooms()`, `withdrawFees()`
- TournamentManager: `setColorNftPouch()`

[I-8] Wrong event emitted in ColorNft.sol:approve()

IMPACT:	Low	**SEVERITY:**	Informational
LIKELIHOOD:	Low	**STATUS:**	Resolved

Target
`ColorNft.sol`

Description
`approve()` emits the Approval event with `msg.sender` as the `owner` argument. Under ERC-721, the Approval event must always use the token's actual owner (`ownerOf(tokenId)`), even when `approve()` is executed by an approved operator (via `isApprovedForAl l`).

Impact
This does not change onchain approval state, but it breaks ERC-721 event semantics and can cause indexers, wallets, and marketplace integrations that rely on events to attribute approvals to the wrong owner or become desynced.

Recommendation
Emit the Approval event with the nft owner, not `msg.sender`