QuillAudits

AUDIT REPORT

February, 2025

For

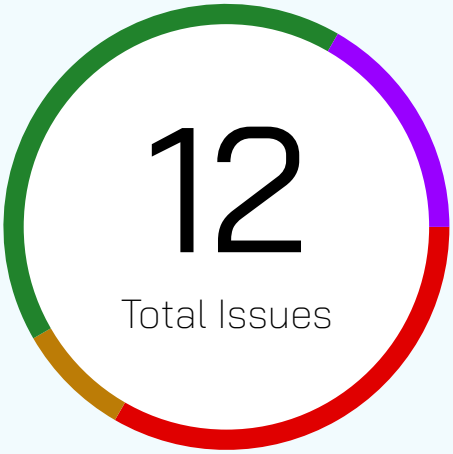
Junky Ursas

Table of Content

Executive Summary

Project name	Junky Ursas
Overview	Junky Ursas is a flagship NFT collection within the Junky Ecosystem, serving as a core component of Junky Bets, a leading GambleFi hub on the Berachain blockchain. It provides exposure to every game developed by the founding team, offering unique features such as LP (Liquidity Provider) Vaults and BGT incentives
Method	Manual Analysis, Functional Testing, Automated Testing
Audit Scope	The scope of this Audit was to analyze the Junky Ursas Smart Contracts for quality, security, and correctness.
Contracts in Scope	https://github.com/Junky-Ursas/JU-contracts/tree/main/pvp_games Branch: main pvp_games/Brumble.sol pvp_games/UrsarollV2.sol
Commit Hash	2620582c3d86ec09668adfca202ce6176369c57f
Language	Solidity
Blockchain	Berachain
Review 1	2nd Jan 2025 - 15th Jan 2025
Updated Code Received	17 th January 2025
Review 2	17th January 2025 - 21st January 2025
Fixed In	d1ea05e9dfbdb87430bed0f3d0519c21060b2f01

Number of Issues per Severity



High	4 (33.33%)
Medium	1 (8.33%)
Low	5 (41.67%)
Informational	2 (16.67%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**4**	**1**	**5**	**2**
Acknowledged	0	0	0	0
Partially Resolved	0	0	0	0

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

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] 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 statistical analysis.

Types of Severity

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

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

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

Resolved

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

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.

High Severity Issues

Legitimate players may be unable to join the game due to incorrect player count after refunds

Resolved

Path

pvp_games/Brumble.sol & pvp_games/UrsarollV2.sol

Function

playBrumble() & playBrumbleZap() & refundUrsaroll()

Description

The playBrumble function uses the playerAddresses.length to determine if the maximum number of players (gameConfig.maxPlayers) has been reached. However, when a player requests a refund via the refundBrumble function, their address is not removed from the playerAddresses array. This creates two problems:
1. Layer count mismatch: The playerAddresses array retains refunded players, artificially inflating its length and blocking legitimate players from joining even when there are available slots.
2. Potential dos vulnerability: A malicious user can repeatedly join the game and request refunds to fill the playerAddresses array, preventing other players from participating.

Recommendation

Modify the refundBrumble function to remove the player's address from the playerAddresses array when they are refunded. This can be done by swapping the refunded player's address with the last address in the array and then reducing the array length by one.

Inability to finalize or end the game due to malicious user contracts reverting prize or refund transfers

Resolved

Path

pvp_games/Brumble.sol, pvp_games/UrsarollV2.sol

Function

finishBrumble() & emergencyEndGame(), finishUrsarollRound() and cancelRound()

Description

1. FinishBrumble function issue: In the finishBrumble function, prizes are distributed to winners using the .call method. If a malicious winner has joined the game using a contract that deliberately reverts on receiving Ether, it will cause the entire for loop to revert, preventing the admin from completing the game.
2. EmergencyEndGame function issue: Similarly, in the emergencyEndGame function, refunds are sent to all participants using .call. If any participant uses a contract that reverts on receiving Ether, the function will fail, leaving the game in an incomplete state.
3. finishUrsarollRound() function issue: In the finishUrsarollRound() function, prizes are distributed to winners using the .call method. If a malicious winner has joined the game using a contract that deliberately reverts on receiving Ether, it will cause the entire for loop to revert, preventing the admin from completing the game. Winner can permanently DOS the protocol from finalizing a round and starting new round
4. cancelRound() function issue: Similarly, in the cancelRound() function, refunds are sent to all participants using .call. If any participant uses a contract that reverts on receiving Ether, the function will fail, leaving the game in an incomplete state. Attackers can prevent the protocol owner from canceling round.

All scenarios create a Denial of Service (DoS) vulnerability that malicious users can exploit to disrupt the game's functionality.

Recommendation

Instead of sending Ether directly to users in the finishBrumble and emergencyEndGame functions, store the amounts owed in a mapping. Users can then claim their prizes or refunds by calling a withdraw function.

Player pays 5% less entry fee in playBrumbleZap, affecting prize pool

Resolved

Path

pvp_games/Brumble.sol

Function

playBrumbleZap()

Description

In the playBrumbleZap function, players can enter the game by paying a fee in native tokens that is equivalent to 95% of the required entry fee, rather than the full amount as required in the playBrumble function. This discrepancy results in players contributing less than the intended entry fee, leading to an unfair prize pool and incorrect accounting within the protocol. The prize pool will be artificially inflated due to the reduced contributions from players using playBrumbleZap. This impacts the fairness and integrity of the game, as the protocol expects all players to contribute an equal amount to the prize pool.

Recommendation

1. Adjust the logic in the playBrumbleZap function to ensure the player contributes the full entry fee amount. Specifically, the required entry fee should be verified against the amount converted through the token swap.
2. Alternatively, consider adjusting the gameConfig.entryFee to account for a token swap mechanism if the discount is intentional, but this should be clearly documented.

Fee from prize pool is being deducted twice in Ursaroll contract

Resolved

Path

pvp_games/UrsarollV2.sol

Function

entropyCallback() & finishUrsarollRound()

Description

In the Ursaroll contract, the fee from the prize pool is deducted twice. The first deduction occurs in the entropyCallback function, where the prize pool for the round is set with a 0.5% fee already subtracted (round.prizePool = round.roundTotalTickets * ticketPrice * 995 / 1000). Then, in the finishUrsaroll-Round function, the fee is deducted again with the calculation uint256 fee = prize * victoryFee / 100000; and sent to the fee recipient via _distributeFees(fee). This results in the winner receiving less than expected, as the fee is being applied twice.

Recommendation

Clearly document the expected behavior of how the prize pool and fee distribution should work in the contract.

Medium Severity Issues

Leftover amount is being sent to fee recipient instead of returning to the user

Resolved

Path

pvp_games/UrsarollV2.sol

Function

playUrsaroll()

Description

In the playUrsaroll function, the leftover amount after the user deposits funds for the rounds is being sent to the fee recipient via _distributeFees. The leftover funds should logically be returned to the user who initiated the transaction. Instead of refunding the excess funds, this implementation funnels them to the fee recipient, which could cause dissatisfaction among users, as they may expect the full amount they sent to be used for the game or refunded if not fully used.

Recommendation

1. Modify the logic so that any leftover amount is returned to the user who called the function, rather than being sent to the fee recipient.
2. Update the function to either send the excess funds back to the user's address or allow the user to specify whether they want to donate the leftover amount to the fee recipient.
3. Document the intended behavior clearly to avoid confusion.

Low Severity Issues

Game id can be changed by the owner during an on-going game, potentially causing failed callbacks

Resolved

Path

pvp_games/Brumble.sol

Function

setGameId()

Description

The setGameId function allows the owner to change the currentGameId at any time, including during an ongoing game. This creates the following issues:

Failed callbacks: If the game relies on the Pyth Entropy contract for randomness or any external oracle service, changing the gameId mid-game will break the linkage between the oracle's response and the ongoing game. This could result in failed callbacks or incorrect randomness being assigned.

Recommendation

Restrict the ability to change the gameId to ensure it cannot be modified during an ongoing game. Implement safeguards to enforce this restriction.

Maxplayers mismatch in gameconfig comment and code

Resolved

Path

pvp_games/Brumble.sol

Function

initialize()

Description

The maxPlayers value in the gameConfig is set to 10, but the comment says “Maximum 100 players allowed.” This discrepancy can confuse developers and users, as the code does not match the comment.

Recommendation

If the intention is to allow 100 players, update the maxPlayers value to 100.

drawWinnerUrsaroll() can be called more than once for a single round

Resolved

Path

pvp_games/UrsarollV2.sol

Function

drawWinnerUrsaroll()

Description

The drawWinnerUrsaroll() allows the owner to request entropy and initiates the process of selecting a winner. However, the function doesn't prevent the owner from calling it again an already initiated round, this also causing the round.sequenceNumber to be changeable. The issue here is in the entropyCallback(), after getting the roundIndex it fails to check if the round.sequenceNumber is still == sequenceNumber provided, as we discussed it is possible for owner to change the sequenceNumber to a new request before the callback.

Recommendation

Check round.sequenceNumber == sequenceNumber

Withdrawal is highly centralized

Resolved

Path

pvp_games/UrsarollV2.sol

Function

withdraw()

Description

High Centralization Risk: Critical withdrawal function with no form of restriction as to what can be withdrawn or when at the dispense of the owner. This poses a risk if the owner's private key is compromised, it could lead to significant disruptions or misuse of the contract.
In recent history, similar centralization vulnerabilities have led to severe consequences, where the compromise of a single private key resulted in the loss of millions of dollars.

Recommendation

There should be a form of restriction as to what can be withdrawn or when, considering the trust on the allocated prizepool and that there could be ongoing rounds. Also use a means of multi-signature wallet for executing critical owner functions. This requires multiple authorized signatures to approve critical actions, reducing the risk of a single point of failure.

Usage of magic number for ticket count validation in Ursaroll contract

Resolved

Path

pvp_games/UrsarollV2.sol

Function

playUrsarollZap()

Description

The playUrsarollZap function uses a hardcoded magic number 70 to validate the maximum number of tickets (count) a user can deposit. Magic numbers in code make it difficult to understand the logic and maintain the contract, as they lack context and are prone to being missed during future updates or modifications.

For better readability and maintainability, the code should use a predefined constant or a configurable variable (e.g., maxFutureRounds) instead of a magic number. This approach also ensures consistency across the contract if the value needs to be updated.

Recommendation

Replace the hardcoded value 70 with a named constant or use the maxFutureRounds variable already defined in the contract.

Informational Severity Issues

Confusing use of maxFutureRounds variable in Ursaroll contract

Resolved

Path

pvp_games/UrsarollV2.sol

Function

N/A

Description

The maxFutureRounds variable in the Ursaroll contract is misleading, as there is no actual maximum round limit enforced in the game. The name maxFutureRounds implies that the game has a cap on the number of future rounds, but this is not the case. In reality, the game allows users to buy a maximum of 68 tickets for the next round, even after the 6169th game is finished. This mismatch between the variable name and its actual functionality could confuse developers and players.

In the current implementation, after finishing the 6169th game, when the finishUrsarollRound() function calls _startNewRound(), the next round's index will be 6170. From this point, users are still able to buy a maximum of 68 tickets for future rounds, despite the variable name suggesting some kind of round limit.

Recommendation

Rename maxFutureRounds to something more representative of its actual purpose, such as maxTicketsForFutureRounds or maxTicketsPerRound, to avoid confusion. This will better reflect the role of the variable in limiting the number of tickets users can purchase rather than capping the number of rounds.

Contract incompatibility with non-standard ERC20 tokens like USDT

Resolved

Path

pvp_games/UrsarollV2.sol

Function

playUrsaroll() & playUrsarollZap() e.t.c

Description

The UrsarollV2 contract uses `IERC20.approve` and `IERC20.transferFrom` to interact with input tokens. However, there are some tokens that don't follow the IERC20 interface, such as USDT. This could lead to transaction reverts when deploying a contract, or depositing tokens.

Recommendation

It's recommended to use functions such as forceApprove and safeTransferFrom from the SafeERC20 library from OpenZeppelin to interact with input tokens.

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 Junky Ursas. We performed our audit according to the procedure described above.

Some issues of High/low/medium and informational severity were found. Some suggestions, gas optimizations and best practices are also provided in order to improve the code quality and security posture.

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 recieved 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 secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

February, 2025

For

Junky Ursas

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com