QuillAudits

AUDIT REPORT

February, 2025

For

Junky Ursas

Table of Content

Table of Content 02

Executive Summary 04

Number of Issues per Severity 06

Checked Vulnerabilities 07

Techniques & Methods 09

Types of Severity 11

Types of Issues 12

High Severity Issues 13

1. Extra value manipulation allows 100% win rate for all iterations 13

2. Lack of token validation in handleDeposit function 14

3. Incorrect validation of minBetAmount in handleDeposit function 15

4. Wrong time check perform on refund function 16

Medium Severity Issues 17

1. Incorrect random value used in applySpecialSymbols function 17

2. handlePayout() approval will fail for some tokens if not approved to zero first 18

3. Division before multiplication 19

Low Severity Issues 20

1. No validation on userRandomNumber and gameAddress 20

2. Incorrect check for number of games played in Junky Slots Super 21

3. Magic number used instead of maxIterations in applySpecialSymbols function 22

4. Hardcoded array index for Multiplier in Junky Slots V2 23

5. House edge set to Zero 24

6. Possible gas limitations due to entropyCallback implementation 25

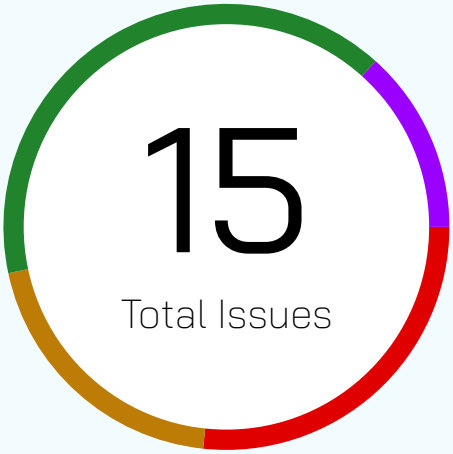
Informational Severity Issues 26

1. Incorrect Error Message for Ether Bet Validation 26

2. Contract incompatibility with non-standard ERC20 tokens like USDT 27

Closing Summary & Disclaimer 28

Number of Issues per Severity



High	4 (26.67%)
Medium	3 (20.00%)
Low	6 (40.00%)
Informational	2 (13.33%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**4**	**3**	**4**	**1**
Acknowledged	0	0	**2**	**1**
Partially Resolved	0	0	0	0

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 Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	The scope of this Audit was to analyze the Junky Ursas Smart Contracts for quality, security, and correctness.
Project URL	https://www.junkyursas.com/
Contracts in Scope	https://github.com/Junky-Ursas/JU-contracts/tree/main/single_games Branch: Main single_games/BlinkoV2.sol single_games/HoneyFlipV2.sol single_games/JunkySlotsSuper.sol single_games/JunkySlotsV2.sol libs/JunkyUrsasEventsLib.sol libs/JunkyUrsasGamesLib.sol libs/JunkyUrsasStructsLib.sol
Commit hash	2620582c3d86ec09668adfca202ce6176369c57f
Language	Solidity
Blockchain	Berachain

Methods	Manual Analysis, Functional Testing, Automated Testing
Review 1	16th Jan 2025 - 29th Jan 2025
Fixed In	f45d1e22722c1515d49ad99c84baa59da7bd619d

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

Extra value manipulation allows 100% win rate for all iterations

Resolved

Path

single_games/HoneyFlipV2.sol & libs/JunkyUrsasGamesLib.sol

Function

playHoneyflip()

Description

In the Honeyflip game, the extra parameter in the GameConfig struct can be manipulated to pass an overly advantageous value (e.g., 100). This causes the maxWagerNotExceeded modifier to bypass appropriate checks, enabling users to win all 69 iterations unfairly. Specifically, when config.extra is set to 100, the calculated wager multiplier becomes invalid and allows users to exploit the game by ensuring all iterations yield a win, resulting in significant losses for the protocol.

Recommendation

1. Add validation logic to ensure that config.extra is within a specific value according to different games, well-defined range (e.g., 1 <= config.extra <= 69).
2. Ensure that the maximum wager calculation cannot be influenced by arbitrary user-defined parameters like extra.

Impact

1. A malicious actor can exploit this loophole to win all iterations in a single game, regardless of randomness or probability.
2. This results in a direct loss of funds from the protocol's treasury or prize pool, jeopardizing the economic security of the system.

Lack of token validation in handleDeposit function

Resolved

Path

libs/JunkyUrsasGamesLib.sol

Function

handleDeposit()

Description

The handleDeposit function does not validate the config.token address passed by the user. As a result, a malicious actor can pass an arbitrary or garbage token address, which could lead to execution being hijacked. For example:
A user could pass a non-standard or malicious ERC20 token address with custom logic in its transferFrom or allowance functions, potentially leading to reentrancy attacks or denial of service.
The contract may also interact with a token that doesn't conform to the ERC20 standard, causing unexpected behavior.
This lack of token validation can compromise the integrity of the game logic and expose the contract to significant risks.

Recommendation

1. Maintain a list of approved tokens and ensure that only these tokens are accepted for deposits.
2. Ensure that the provided token address is a valid ERC20 token by checking the response of critical ERC20 functions (allowance, transferFrom, etc.).

Impact

1. Reentrancy Attack: Malicious tokens with custom transferFrom logic can exploit the function to execute arbitrary code.
2. Denial of Service: Non-standard or broken tokens may cause the function to revert, blocking users from playing the game.
3. Protocol Exploit: An invalid token could hijack the game logic, leading to erroneous payouts or treasury depletion.

Incorrect validation of minBetAmount in handleDeposit function

Resolved

Path

libs/JunkyUrsasGamesLib.sol

Function

handleDeposit()

Description

The handleDeposit function checks if the total wager (calculated as config.wager * config.count) is greater than or equal to the minBetAmount. This approach allows users to bypass the intended minimum bet requirement by manipulating the wager and count values. For instance:
A user can set config.wager to 0.005 ether and config.count to 3, resulting in a totalWager of 0.015 ether, which satisfies the condition (0.015 ether >= minBetAmount).
However, the intended behavior is that the minimum wager per game should be enforced, not just the total wager. This allows users to exploit the system by paying far less than the expected minimum for each round.

Recommendation

1. Modify the validation logic to ensure that the config.wager (wager per round) meets the minBetAmount threshold.
2. Ensure that the total wager (config.wager * config.count) is appropriately calculated and validated based on the context.

Impact

1. Users can bypass the minimum wager requirement for individual games, resulting in unfair play and reduced prize pools.
2. A malicious actor can abuse this flaw to play multiple rounds at significantly reduced costs, undermining the game's economic balance and potentially draining the contract

Wrong time check perform on refund function

Resolved

Path

libs/JunkyUrsasGamesLib.sol

Function

refund()

Description

It was intended that after the entropy callback allowed time (resolve period) has expired and no longer executable can the refund be called, however was implemented in the opposite way due to wrong operator used in checking resolve period has expired

require(config.timestamp + 1000 > block.timestamp, "Resolve period is not over yet");

Instead of requiring the resolve period be over before refund can be called, this requires the resolve period to not be over.

Recommendation

Change “>” to “<”

require(config.timestamp + 1000 < block.timestamp, "Resolve period is not over yet");

Medium Severity Issues

Incorrect random value used in applySpecialSymbols function

Resolved

Path

single_games/JunkySlotsSuper.sol

Function

calculateGameLogic()

Description

In the calculateGameLogic function, the applySpecialSymbols function is called using currentRandom shifted by bitsUsed. However, a new random value, rndNext, is already generated and should be used instead for better randomness propagation.

Recommendation

Use rndNext for the applySpecialSymbols function to ensure proper randomness

Impact

Using an outdated random value (currentRandom) instead of the newly generated one (rndNext) reduces the randomness of the results, potentially making outcomes predictable.

handlePayout() approval will fail for some tokens if not approved to zero first

Resolved

Path

single_games/JunkySlotsSuper.sol

Function

handlePayout()

Description

When approving some tokens e.g USDT, the allowance value needs to be set to 0 first before it can be used correctly else it will revert. This will result in the contract being unable to execute the game again since handlePayout() which this is performed in, is called in the entropyCallback().

Recommendation

Use forceApprove

Division before multiplication

Resolved

Path

single_games/JunkySlotsSuper.sol

Function

getMaxWinPayout(), getMaxTotalWager()

Description

In Solidity, performing division before multiplication can lead to a loss of precision due to integer division truncation. This means that the fractional part of the result is truncated, leading to inaccurate calculations.

Recommendation

Change the order of calculation to multiply before division.

Low Severity Issues

No validation on userRandomNumber and gameAddress

Resolved

Path

libs/JunkyUrsasGamesLib.sol

Function

playGame()

Description

The playHoneyflip function does not perform validation on the userRandomNumber and gameAddress parameters in the GameConfig struct. This allows users to pass arbitrary values for userRandomNumber, potentially impacting the fairness or predictability of game outcomes.

Recommendation

Ensure that the userRandomNumber and gameAddress parameters pass some validation to prevent malicious or repetitive values.

Incorrect check for number of games played in Junky Slots Super

Resolved

Path

single_games/JunkySlotsSuper.sol

Function

calculateGameLogic()

Description

The calculateGameLogic function uses the variable i to compare against maxIterations when it should use the playedCount variable. This could lead to unintended behavior, such as:
1. Exceeding the maximum allowed iterations (maxIterations).
2. Incorrectly calculating game outcomes based on the wrong loop condition.

Recommendation

Use flags.playedCount instead of i to ensure the number of games played is consistent with the recorded count.

Magic number used instead of maxIterations in applySpecialSymbols function

Resolved

Path

single_games/JunkySlotsSuper.sol

Function

calculateGameLogic()

Description

The applySpecialSymbols function uses a hardcoded magic number (70) as the upper limit for spinIndex. This is not ideal because:
1. It reduces code readability and flexibility.
2. Future changes to the maximum iterations (maxIterations) will not automatically propagate to this function, increasing the risk of inconsistencies.

Recommendation

Replace the magic number with the configurable maxIterations variable.

Hardcoded array index for Multiplier in Junky Slots V2

Resolved

Path

single_games/JunkySlotsV2.sol

Function

playJunkySlotsV2()

Description

In the playJunkySlotsV2 function, the maxWagerNotExceeded modifier uses a hardcoded array index (junkySlotsV2Multipliers[0]), which represents a multiplier of 27. However, the junkySlotsV2Special-Multipliers array defines possible multipliers as [69, 99, 36], which are not being used.
This hardcoding restricts flexibility and does not align with the intended game logic, where multipliers can vary.
Using only 27 as the multiplier can mislead users and developers about the actual game mechanics and its maximum payout potential.

Recommendation

Replace the hardcoded array index with dynamic logic that allows selection of appropriate multipliers based on game configuration or conditions.

House edge set to Zero

Acknowledged

Path

libs/JunkyUrsasGamesLib.sol

Function

initialize()

Description

The houseEdge variable, which determines the percentage of wagers retained by the contract as profit, is set to 0. This means the platform operator will not make any revenue from the games, which could be intentional for testing but is not suitable for a live environment.
Without a house edge, the platform cannot sustain operational costs or ensure long-term profitability.

Recommendation

Set the houseEdge to an appropriate percentage value based on the game's profitability model (e.g., 500 for a 5% house edge).

Possible gas limitations due to entropyCallback implementation

Acknowledged

Path

libs/JunkyUrsasGamesLib.sol

Function

entropyCallback()

Description

According to pyth network, calculateGameLogic() which involves complex iterations and handle-Payout() should not be done in the entropyCallback() due to gas limits. see: https://docs.pyth.net-work/entropy/best-practices?ref=pyth-network.ghost.io#limit-gas-usage-on-the-callback.

Recommendation

Consider storing the random number received from the callback and continue the flow on a different function.

Informational Severity Issues

Incorrect Error Message for Ether Bet Validation

Resolved

Path

libs/JunkyUrsasGamesLib.sol

Function

handleDeposit()

Description

the error message for Ether bet validation (require(msgValue > fee, "Bet amount too low");) is misleading. The condition checks if the value sent (msgValue) is greater than the fee, which is unrelated to the bet amount.

Recommendation

Update the error message to correctly describe the condition being checked.

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