QuillAudits

AUDIT REPORT

February, 2025

For

Junky Ursas

Table of Content

Table of Content 02

Executive Summary 03

Number of Issues per Severity 05

Checked Vulnerabilities 06

Techniques & Methods 08

Types of Severity 10

Types of Issues 11

High Severity Issues 12

1. Double Accounting 12

2. validateAddLiquidityInputs reverts for native token deposits 13

Medium Severity Issues 14

1. Division by zero in calculateVaultTokens() 14

2. addLiquidity() has no slippage protection 15

3. calculateWithdrawalAmount returns 0 for small amounts 16

Low Severity Issues 17

1. getAllVaults() reverts even after creating vaults 17

Informational Severity Issues 18

1. No check for zero ETH deposits in depositETH 18

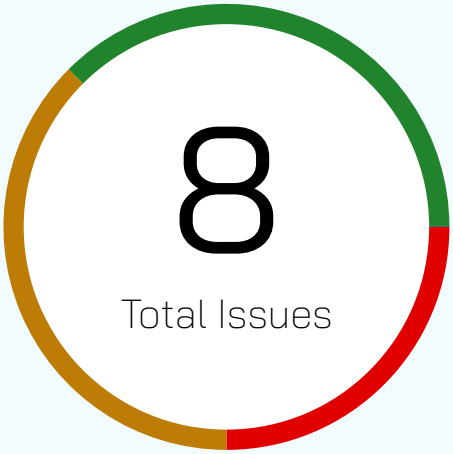
2. Unused variables liqTokenAddress & vaultManager 19

Closing Summary & Disclaimer 20

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.
Project URL	https://www.junkyursas.com/
Contracts in Scope	https://github.com/Junky-Ursas/JU-contracts/tree/main/service Branch: main service/VaultManagerV2.sol service/VaultTokenV2.sol service/Bankroll.sol service/ERC20JU.sol MVP/AA_MVP.sol
Commit Hash	2620582c3d86ec09668adfca202ce6176369c57f
Language	Solidity
Blockchain	Berachain
Review 1	30th Jan 2025 - 10th Feb 2025
Fixed In	f45d1e22722c1515d49ad99c84baa59da7bd619d

Number of Issues per Severity



High	2 (25.00%)
Medium	3 (37.50%)
Low	3 (37.50%)
Informational	0 (0.00%)

Severity

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

Double Accounting

Resolved

Path

service/VaultManagerV2.sol

Function

updateAverageExchangeRate()

Description

When a user adds liquidity to a vault, it mints vault tokens (vaultTokens) to the user through the mintVaultTokens() function, which also updates the user total tokens with the new added token:

(user.deposit.vaultTokens += vaultTokens)

and then updates the average exchange rate for the user in the updateAverageExchangeRate() using the user totalAmount and totalTokens, in the function this totalTokens is calculated like so:

totalTokens = userDeposit.vaultTokens + vaultTokens

As you can see, it still adds the vaultTokens already added in mintVaultTokens(), however it still accounts again when getting the totalTokens like it hasn't been added. This will impact the user rate accounting

Recommendation

Update totalTokens to just userDeposit.vaultTokens

validateAddLiquidityInputs reverts for native token deposits

Resolved

Path

service/VaultManagerV2.sol

Function

validateAddLiquidityInputs

Description

The function validateAddLiquidityInputs performs a strict equality check (msg.value == amount) when the liquidity token is ETH (native token). This can cause the function to revert unexpectedly when users deposit ETH, even if they send a correct or slightly higher amount (e.g., to cover additional fees).

Recommendation

Instead of using msg.value == amount, allow the user to send at least the required amount (msg.value >= amount), and if there's an excess, the difference can be refunded.

Medium Severity Issues

Division by zero in calculateVaultTokens()

Acknowledged

Path

service/VaultManagerV2.sol

Function

addLiquidity()

Description

In the getVaultTokenPrice(), if bankrollBalance == 0 it returns 0, according to the comment this is to avoid division by zero, however this function is still used in calculateVaultTokens() where returning 0 would still cause a division by zero.
(amount * 100000) / getVaultTokenPrice(liqTokenAddress);

Recommendation

Division by zero should be addressed in calculateVaultTokens() as well. Note: getVaultTokenPrice() should remain unchanged

addLiquidity() has no slippage protection

Resolved

Path

service/VaultManagerV2.sol

Function

addLiquidity()

Description

Shares returned in addLiquidity() depends on the calculation return in getVaultTokenPrice(), the calculation is based on:
(bankrollBalance * 100000) / totalVaultTokens;
Any changes in these values through frontrunning and/or market changes that take effect before the user's transaction to add liquidity will affect the shares returned.
However, the function lacks slippage protection, which is critical to safeguarding users' assets from frontrunning attacks and market volatility.

Recommendation

Implement slippage protection in the function

calculateWithdrawalAmount returns 0 for small amounts

Resolved

Path

service/VaultManagerV2.sol

Function

calculateWithdrawalAmount

Description

The function calculateWithdrawalAmount fails to properly compute withdrawal amounts for small values (e.g., 1 billion wei or less). This can cause:
1. Improper updates to totalBalance and totalWithdrawn in both vaults and userInfo structs.
2. Zero-value withdrawals, where the user gets nothing, but their vaultTokens are still reduced

Recommendation

Increase precision to avoid rounding to zero.

Low Severity Issues

getAllVaults() reverts even after creating vaults

Resolved

Path

service/VaultManagerV2.sol

Function

getAllVaults

Description

The function getAllVaults() is expected to return an array of all vaults with their updated prices, but it reverts every time, even after vaults have been created.

Recommendation

Use correct indexing by ensuring that the vault lookup uses the correct address key

Informational Severity Issues

No check for zero ETH deposits in depositETH

Resolved

Path

service/Bankroll.sol

Function

depositETH()

Description

The depositETH function does not explicitly check whether the deposited amount is greater than zero. Although require(msg.value >= amount, "Invalid ETH amount"); ensures that the sent ETH is at least equal to amount, it does not prevent cases where a user accidentally or intentionally deposits 0 ETH. Allowing zero-value deposits could lead to unnecessary event emissions, increased gas usage, or potential edge-case issues.

Recommendation

Add an explicit check to prevent zero-value deposits.
Add an explicit check to prevent zero-value deposits.

Unused variables liqTokenAddress & vaultManager

Resolved

Path

service/ERC20JU.sol

Function

N/A

Description

The variables liqTokenAddress and vaultManager are declared but never used in ERC20JU.sol. Keeping unused state variables increases gas costs, makes the contract less readable, and may indicate incomplete or unnecessary code.

Recommendation

If these variables are not required for any logic, they should be removed to optimize storage and gas usage.

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