QuillAudits

Audit Report
August, 2024

For

Table of Content

Table of Content

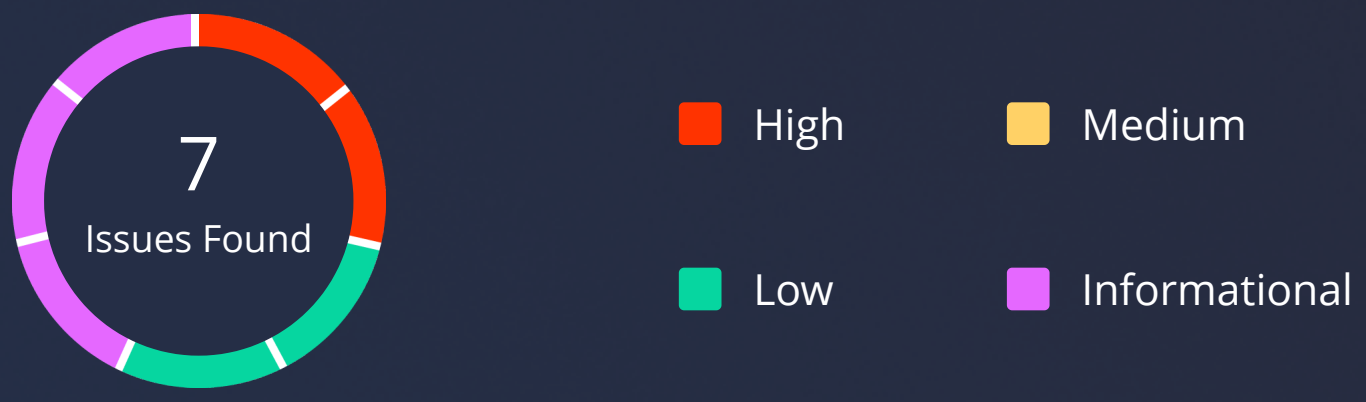
Executive Summary

Project Name	AntiGravity
Overview	The project involves a system of interconnected smart contracts designed to manage and distribute NFTs (FuelCells) and related rewards. The FuelCells contract is an ERC721 NFT contract that allows the controlled minting and burning of NFTs, with permissions granted to the LaunchControlCenter to handle minting during active phases. The LaunchControlCenter is responsible for processing NFT purchases and splitting payments into predefined streams, while the Jackpot and Treasury contracts manage the distribution of lottery winnings and yield, respectively, based on the NFTs minted in different journeys.
Timeline	9th August 2024 - 16th August 2024
Updated Code Received	27th August 2024
Second Review	27th August 2024
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 Antigravity Contracts for quality, security, and correctness.
Source Code	https://github.com/chain-labs/antigravity-core/commit/26d3bcc719293a720d241f6f55f8cc94c5c74fb3

Executive Summary

Contracts In-Scope	EvilAddress/EvilAddress.sol EvilAddress/EvilAddressStorage.sol FuelCell/FuelCell.sol FuelCell/FuelCellStorage.sol Jackpot/Jackpot.sol Jackpot/JackpotStorage.sol JourneyPhaseManager/JourneyPhaseManager.sol JourneyPhaseManager/JourneyPhaseManagerStorage.sol LaunchControlCenter/LaunchControlCenter.sol LaunchControlCenter/LaunchControlCenterStorage.sol Treasury/Treasury.sol Treasury/TreasuryStorage.sol constants/ERA3Constants.sol
Branch	26d3bcc719293a720d241f6f55f8cc94c5c74fb3
Fixed In	950c2b33eb72eecfe7730ee53bc7824e60984dff

Number of Security Issues per Severity



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

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- 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's 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

Hardhat, Foundry.

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

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

These are the issues identified in the initial audit and have been successfully fixed.

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

1. Locked Tokens in Jackpot Contract Due to Incorrect totalPayout Calculation

Path

Jackpot.sol

Function

announceLotteryResult

Description

The Jackpot contract incorrectly handles the totalPayout calculation, leading to locked tokens when users claim rewards from previous lotteries. When a user claims their reward, the totalPayout is not diminished, resulting in an inaccurate calculation of the available balance for subsequent lotteries.

Scenario:

1. Lottery 1: The contract has 100 coins, all of which are set to be distributed.
2. The totalPayout is recorded as 100 coins.
3. The contract balance is 100 coins.
4. A user claims 10 coins from Lottery 1, reducing the balance to 90 coins.
5. Another 100 coins are added to the contract for future rewards.
6. Lottery 2 is set up, but the payout calculation results in only 90 coins being available for distribution instead of the expected 100 coins due to the formula:

   darkToken.balanceOf(address(this)) - totalPayout;

   This discrepancy occurs because the totalPayout was not reduced when the user claimed their 10 coins from Lottery 1, leaving 10 tokens locked in the contract.

Recommendation

To prevent the locking of tokens, the totalPayout variable should be replaced with a pendingPayout that accurately tracks the remaining payouts for each lottery. This pendingPayout should be diminished each time a user claims their reward, ensuring that the available balance for future lotteries is correctly calculated.

Status

Resolved

2. Potential for Incorrect Yield Calculation Due to Balance Change in distributeYield Function

Path

Treasury.sol

Function

distributeYield

Description

The distributeYield function in the contract relies on calculating the available yield based on the darkToken balance at the time of execution, subtracting the totalYieldDistributed. However, this approach is flawed because the darkToken balance can change at any time, particularly if a user claims rewards from the treasury while the function is being executed. This can lead to an inflated totalYieldDistributed value, causing the contract to distribute incorrect amounts of yield across different journeys. As a result, some tokens may be locked or misallocated, disrupting the overall accounting and fairness of the distribution process.

Recommendation

To prevent these issues, the yield calculation should be insulated from any balance changes that might occur due to user actions, such as unwrapping (claiming) rewards from the treasury.

Status

Resolved

Low Severity Issues

3. Sequential Journey Assumptions Lead to Bonus Locking and Improper Accounting

Path

Jackpot.sol

Function

announceLotteryResult

Description

The Jackpot contract currently assumes that journeys are sequential and that each journey's bonuses are managed within the same contract balance. However, this approach does not account for the possibility of skipped journeys due to bot malfunctions, which can result in locked bonuses and improper accounting. Additionally, managing all journeys within a single allocation pool can lead to inconsistencies and difficulties in tracking bonuses and payouts for individual journeys.

Scenario:

1. Journey 1 is processed normally.
2. The Evil contract adds a bonus for Journey 2.
3. Due to a malfunction in the bot, Journey 2 does not have its rewards claimed.
4. As a result, the bonus for Journey 2 remains locked and unclaimed.
5. When the bot processes rewards for Journey 3, it does not account for the
6. unclaimed rewards from Journey 2, leading to potential errors where users could claim rewards from both Journey 2 and Journey 3.

This scenario results in:

- **Locked Bonus:** Bonuses for skipped journeys remain locked and cannot be properly allocated.
- **Improper Accounting:** Users might be able to claim rewards for multiple journeys when only one journey's rewards should be available, distorting the reward distribution.
- **Inconsistencies:** Managing all journeys within a single allocation pool can lead to inconsistencies and challenges in accurately tracking and distributing bonuses for individual journeys.

Recommendation

Introduce mechanisms to handle non-sequential journeys and address issues caused by bot malfunctions. Additionally, implement separate allocation pools for each journey to ensure accurate accounting and distribution of bonuses.

Status

Resolved

4. Incorrect Calculation of totalPausedTime in JourneyPhaseManager

Path

JourneyPhaseManager.sol

Function

unpause

Description

The current implementation of the unpause() function in the JourneyPhaseManager contract only accounts for the duration of the most recent pause when calculating totalPausedTime. This leads to an inaccurate reduction in journey time, as only the last pause is considered. If the contract is paused multiple times, the cumulative paused duration is not properly captured, potentially resulting in journeys not being extended by the correct amount of time.

Recommendation

To accurately account for all pause intervals, the totalPausedTime should accumulate the durations of all pauses

Status

Resolved

Informational Issues

5. Missing Event Emissions in Key Functions

Path

*

Function

*

Description

Several critical functions in the smart contracts are missing event emissions, which impacts transparency and traceability of state changes. Event emissions are crucial for providing clear, on-chain notifications of significant contract updates and actions. The following functions need to be updated to emit events:

1. updateSplits, updateTreasuryAddress, updateJackpotAddress, updateTeamAddress
2. in LaunchControlCenter.
   evilPrune in EvilAddress.sol.
3. updateLauncherAddress in FuelCell.sol.

Without these events, it is challenging for users to track changes and ensure the integrity of contract operations.

Recommendation

Add event emissions to the specified functions to improve transparency and provide clear notifications of changes.

Status

Resolved

6. Lack of Two-Step Ownership Transfer Poses Security Risk

Path

*

Function

*

Description

The current implementation of the ownership transfer process in the smart contracts does not include a two-step confirmation mechanism. This absence increases the risk of accidental or malicious transfers, as the ownership can be transferred with a single transaction.

A two-step ownership transfer is a widely recognized security practice that requires the new owner to explicitly accept ownership before the transfer is finalized. This approach reduces the risk of errors or unauthorized ownership transfers, providing an additional layer of security and ensuring that ownership changes are intentional and verified.

Recommendation

Implement a two-step ownership transfer process in all contracts that involve ownership changes.

Status

Resolved

7. Undocumented Features in Smart Contracts

Path

*

Function

*

Description

During a review of the smart contracts, it was identified that several features are implemented in the code but are not mentioned or described in the project documentation. This lack of documentation can lead to misunderstandings, misconfigurations, and potential misuse of these features. The undocumented features identified are as follows:

1. **Rapture in JourneyPhaseManager:**

   The JourneyPhaseManager contract includes a feature called “Rapture” (isRapturePossible). This functionality appears to have a significant impact on the management of journeys, potentially altering their state or outcomes. However, without documentation, it’s unclear what the exact purpose and effect of this feature are, and how it should be used or configured.

2. **Bonus in Jackpot:**

   The Jackpot contract has a “Bonus” feature that is used in conjunction with lotteries. This bonus system seems to affect the distribution of rewards across different lotteries within a journey. The lack of documentation on how bonuses are calculated, applied, and managed can lead to incorrect assumptions about how the reward distribution works, which might cause issues in reward allocation and user satisfaction.

3. **EvilContract:**

   The EvilContract is referenced and appears to have interactions with other contracts, possibly affecting their state or behavior. The functionality and purpose of this contract are not documented, making it difficult to understand its role within the broader system. This could lead to security vulnerabilities or unintended interactions if the contract’s behavior is not fully understood.

Recommendation

Update the project documentation to include detailed descriptions of the Rapture, Bonus, and EvilContract features.

Status

Resolved

Functional Tests Cases

FuelCells (ERC721)

1. **Minting Functionality**
   - Verify that NFTs can only be minted when the minting phase is active.
   - Ensure that only the LaunchControlCenter contract can mint NFTs.
   - Test that the owner can update the baseURI.
   - Confirm that the owner can change the LaunchControlCenter contract address.
2. **Burning Functionality**
   - Verify that users can burn their NFTs.
   - Ensure that burning an NFT triggers the release of unclaimed lotteries and yield from the Jackpot and Treasury contracts.
   - Confirm that the total number of NFTs minted in a journey is decremented when an NFT is burned.
3. **Interaction with JourneyPhaseManager**
   - Test that the FuelCells contract updates the JourneyPhaseManager with the number of NFTs minted.
   - Verify that the contract correctly queries the current journey and phase from the JourneyPhaseManager.

LaunchControlCenter (Controller)

1. **Minting and Payment Splitting**
   - Ensure that an NFT is minted when a user purchases it using 1 Dark token.
   - Verify that the Dark token is split into the three streams correctly (3% to team, 20% to Jackpot, 77% to Treasury).
   - Test that the LaunchControlCenter does not retain any tokens after the transaction.
2. **Contract Address Management**
   - Verify that the owner can update the contract addresses for FuelCells, Jackpot, and Treasury.
   - Confirm that the owner can change the percentage allocations for the payment streams.

Jackpot

1. **Lottery Management**
   - Test that the Jackpot contract tracks the Dark tokens received for each journey.
   - Verify that the contract identifies the current journey using the JourneyPhaseManager.
   - Confirm that the lottery winner list can be posted by the off-chain keeper using a Merkle tree.
   - Ensure that only eligible participants (those who minted in the current journey and haven't won before) can win.
2. **Funds Disbursement**
   - Verify that the total available funds are correctly calculated and allocated over 10 drawings.
   - Test that the distribution mechanism allows winners to claim their winnings (pull-type payment).
   - Confirm that the correct number of winners is selected per drawing, especially in cases where there are extra winners.

Treasury

1. **Yield Distribution**
   - Test that the yield is distributed according to the geometric sequence formula.
   - Verify that the yield distribution decreases correctly with each journey.
   - Ensure that users can claim their yield after each journey (pull-type payment).
   - Confirm that the Treasury correctly tracks the Dark tokens received for each journey.

JourneyPhaseManager

1. **Journey Tracking**
   - Verify that the contract correctly tracks the start time and duration of each journey.
   - Ensure that the journey can be paused and resumed, with the elapsed time adjusted accordingly.
   - Test that the FuelCells contract can update the number of NFTs minted for each journey.
2. **Automatic Progression**
   - Confirm that the journey progression occurs automatically based on the set start time.
   - Test that the owner can correctly set the start time of the first journey during deployment.

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 the Antigravity codebase. We performed our audit according to the procedure described above.

Some issues of Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Antigravity smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Antigravity smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Antigravity to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
August, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com