QuillAudits

Audit Report
November, 2024

For

Table of Content

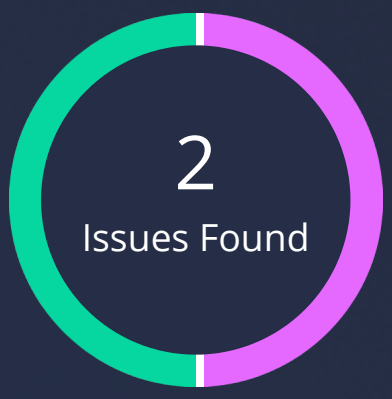
Executive Summary

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.
Audit Scope	https://github.com/chain-labs/antigravity-core/compare/26d3bcc7...dde42b2
Contracts In-Scope	## Changed Files: 1. src/ERA2/DarkClaims.sol 2. src/ERA3/EvilAddress/EvilAddress.sol 3. src/ERA3/EvilAddress/EvilAddressStorage.sol 4. src/ERA3/FuelCell/FuelCell.sol 5. src/ERA3/FuelCell/FuelCellStorage.sol 6. src/ERA3/Jackpot/Jackpot.sol 7. src/ERA3/Jackpot/JackpotStorage.sol 8. src/ERA3/JourneyPhaseManager/JourneyPhaseManager.sol 9. src/ERA3/LaunchControlCenter/LaunchControlCenter.sol 10. src/ERA3/LaunchControlCenter/LaunchControlCenterStorage.sol 11. src/ERA3/Treasury/Treasury.sol 12. src/ERA3/Treasury/TreasuryStorage.sol 13. src/ERA3/constants/ERA3Constants.sol
Commit Hash	4c0c9670a139af695d27f27a903cb2b9d2ce7578
Language	Solidity
Blockchain	EVM

Executive Summary

Method	Manual Analysis, Functional Testing, Automated Testing
First Review	23th October 2024 - 30th October 2024
Updated Code Received	9th November 2024
Second Review	18th Novemeber 2024 - 19th November 2024
Fixed In	415a20c 9fa8308

Number of Security Issues per Severity



High
Medium
Low
Informational

	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	0	0	1	1

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic 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

Checked Vulnerabilities

- Using inline assembly
- Unsafe type inference
- Style guide violation
- 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 statistic 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

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.

Low Severity Issues

1. Mismatch between expected gas and required gas

Path

EvilAddress.sol

Function

evilCall()

Description

The following function is preserving 1500 gas as a buffer to ensure subsequent call is being executed. However comments mention it as 2500 gas instead.

```
function evilCall(address to, uint256 value, bytes memory data, uint256 txGas)
    external
    payable
    onlyOwner
    returns (bool success)
    {
    success = execute(to, value, data, Enum.Operation.Call, txGas == 0 ? (gasleft() - 1500) :
txGas);
    if (!success) {
        revert EvilCallFailed();
    }
    }
```

However, hardcoding gas values can be risky, as Ethereum's gas usage may vary across different versions or updates.
A safer alternative might be to set the gas buffer as a configurable variable, allowing adjustments without contract redeployment.

Recommendation

Synchronize comments and syntax. For removing hardcoding gas dependencies :

define a configurable gas buffer as a variable in the contract, such as gasBuffer, that the owner or an admin can adjust.

```
uint256 public gasBuffer = 1500;

function setGasBuffer(uint256 newBuffer) external onlyOwner {
    gasBuffer = newBuffer;
}
```

In evilCall, you'd then replace the hardcoded value with gasBuffer:

```
success = execute(to, value, data, Enum.Operation.Call, txGas == 0 ? (gasleft() - gasBuffer) : txGas);
```

Status

Resolved

Informational Issues

2. Allowing arbitrary function call might pose a threat if owner is compromised.

Path

EvilAddress.sol

Function

evilCall()

Description

Allowing arbitrary function calls through evilCall pose a security risk if the owner's account is compromised. In such a case, a malicious actor gaining control of the owner's account could potentially call evilCall to execute any arbitrary function on other contracts or even manipulate the contract itself.

Recommendation

Add a time-lock for high-risk functions, requiring a delay before evilCall is executed. This would allow time to detect and respond to potential attacks.

```
mapping(bytes32 => uint256) public callTimestamps;

function requestCall(address to, uint256 value, bytes memory data) external onlyOwner {
    bytes32 callHash = keccak256(abi.encodePacked(to, value, data));
    callTimestamps[callHash] = block.timestamp + delayTime;
}

function executeCall(address to, uint256 value, bytes memory data) external onlyOwner {
    bytes32 callHash = keccak256(abi.encodePacked(to, value, data));
    require(block.timestamp >= callTimestamps[callHash], "Call time-locked");
    delete callTimestamps[callHash];
    // Execute call
}
```

Alternatively, OpenZeppelin's **TimelockController** can also be used

Status

Resolved

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 AntiGravity. We performed our audit according to the procedure described above.

One issue of Low and one issue informational severity were found. Some suggestions, gas optimizations and best practices are also provided in order to improve the code quality and security posture. In The End, Antigravity Team Resolved Both Issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in AntiGravity. 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. 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 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
November, 2024

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com