QuillAudits

Audit Report
November, 2024

For

Quranium

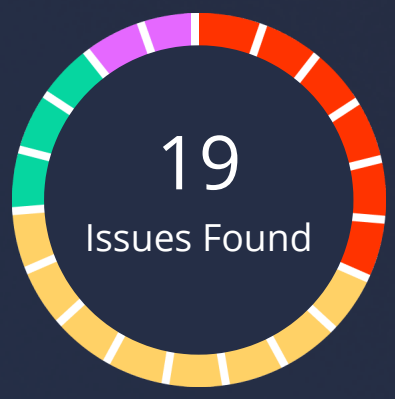
Table of Content

Table of Content

Executive Summary

Project Name	Quranium
Project URL	https://quranium.org/
Overview	Quranium is a Node sales contract, as part of its initiative to develop a quantum-resistant blockchain infrastructure. Key Features of the Node Pre-sale 1. Tiered Node Access: Participants can choose from multiple entry levels, allowing for tailored engagement and corresponding rewards. 2. Priority Whitelisting: Active community members gain early access, enhancing their chances to contribute significantly to the network. 3. Exclusive Rewards: Node buyers can refer new buyers to get discounts on their purchases.
Audit Scope	The Scope of the Audit is to Analyze the Quranium Smart Contract for Security,Code Quality and Correctness.
Contracts In-Scope	https://arbiscan.io/address/0x3A5DfF1Badc2D23eC45fF19526aD7F0183557e75#code
Language	Solidity
Blockchain	Arbitrium
Method	Manual Analysis, Functional Testing, Automated Testing
First Review	15th November 2024 - 18th November 2024
Updated Code Received	27th November 2024
Second Review	27th November 2024 - 29th November 2024
Fixed In	https://drive.google.com/file/d/1zBoKvtCXvWfngXxQ61dxRpAxZyy9BhpN/view?usp=share_link

Number of Security Issues per Severity



High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	0	2
Partially Resolved Issues	0	0	1	0
Resolved Issues	6	7	2	0

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
- Style guide violation
- 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 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.

High Severity Issues

1. tierPrices and tierSizes cannot be updated

Description

The contract utilizes two mappings to handle the TierSizes and TierPrices to handle the current Tier size limit and price.
The current MAX_TIERS is set to 77, however this value can be updated using the `setMaxTiers()`, but this function only changes the Max possible number of tiers. It does not update the necessary values for both mappings.
Hence when adding more Tiers, these Tiers will have no price or size, thereby making it impossible to actually add Tiers.

Recommendation

Add a function that would enable addition of details to each Tier added. This function can be merged with the `setMaxTiers()` or could be a standalone function that looks like this:

```solidity
    function updateTierDetail(uint256 Tier, uint256 price, uint256 size) external onlyOwner {
     require(Tier > currentTier && Tier <= MAX_TIERS, " tier must be greater than the current tier ");
    tierSizes[Tier] = size;
    tierPrices[Tier] = price;
  }
```

Status

Resolved

2. Tier shift would still use old tier price in calculation

Description

The QNodeKey contract automatically switches tiers once one tier is filled up using ``_advanceTier()``. This resets the tier counter, and change
the current Tier to the next one, which is always more expensive and might have a smaller size.
However when this change occurs, either in between an order (A buyNode function call that will finish a current tier and start a new one) or at the start of a new order(A buyNode function call that is starting a new Tier apart from Tier 1), the `buyNode()` function uses the older price when calculating the price addition of the new Tier. This is because the price of the previous Tier has been cached in `pricePerNode` which is then used through the whole calculation.

```solidity
    uint256 totalPrice = 0;
    uint256 remainingQuantity = quantity;
    uint256 referrerReward = 0;
    uint256 pricePerNode = getTokenPrice(paymentToken);// STEP 1: Current price cached here
    if (bytes(referrerCode).length > 0) {
        address referrer = referralCodeToAddress[referrerCode];
        require(referrer != address(0), “Invalid referral code”);
        require(referrer != msg.sender, “Cannot refer yourself”);
        while (remainingQuantity > 0) {

            uint256 discountPerNode = (pricePerNode * 5) / 100;
            uint256 rewardPerNode = (pricePerNode * 20) / 100;
            uint256 availableInCurrentTier = tierSizes[currentTier] - tierMinted;

            if (remainingQuantity <= availableInCurrentTier) {
                totalPrice += (pricePerNode - discountPerNode) * remainingQuantity;//STEP 3: OLD Price still used after step 2
                referrerReward += rewardPerNode * remainingQuantity;
                tierMinted += remainingQuantity;
                remainingQuantity = 0;
            } else {
```

```
            totalPrice += (pricePerNode - discountPerNode) * availableInCurrentTier;
             referrerReward += rewardPerNode * availableInCurrentTier;
             remainingQuantity -= availableInCurrentTier;
             _advanceTier(); // STEP 2: Tier is changed
          }
       }
```

Recommendation

Always refresh the value of `pricePerNode` within the while Loop.

```solidity
    while (remainingQuantity > 0) {
            uint256 pricePerNode = getTokenPrice(paymentToken);
```

Also NOTE that this issue is present when there is no `referralReward` too. The price should be fetched inside its own while block as well.

Status

Resolved

3. Reward does not track currency

Description

The `referralRewards` being used to accumulate rewards for users has a critical bug rendering it useless.
The mapping adds all reward cuts directly, It does not handle the different tokens separately.
Take for example: User uses a referrerCode and purchases 5 NFTs with eth, This records 70000000000000000 which is 0.07 ether as rewards for the referrer. When another user comes to purchase 5 more NFTs in tier one, using that same referrer's code but purchased with USDC, the function adds 215803115 to that referrers balance, which is 215 USDC.
But now, the referrers reward is now `70000000215803115`, which cannot be valued in usdc or ether or wbtc.

Recommendation

I recommend fully calculating the price in ETH first, making the necessary deductions, before now converting price to the different token, and then transfer.
Alternatively, we could have a 3 way mapping for the referral rewards. Referrer => token => amount.

Status

Resolved

4. Buying reverts for ERC20-referral-buys

Description

In the `buyNode` function, when handling purchases of nodes that have referrals, the function accurately calculates the amount the referrer should get.
However, it tries to do the token transfer to the referrer immediately,without first transferring in rewards from the buyer. This causes the function to revert because the contract does not yet have the tokens as at the point when the transfer is being attempted. Hence, for the first transaction for a token(usdt/usdc/wbtc) on the contract , and everytime the admin withdraws token funds from the contract, this function call will revert.

Recommendation

Since `withdrawReferralReward()` has been added, the `buyNode()` function can completely exclude the transfer of rewards to the recipient and let recipients withdraw their own rewards later.

On the other hand, if rewards are to be handled by in `buyNode()` then transfer the rewards at the end of the buyNode function, after transferring in tokens from buyers

Status

Resolved

5. New Tier uses old tier price in _simulatePurchase.

Description

When _simulatePurchase() is called the pricePerNode variable doesn't get updated after the current tier's slot is completely filled. As remainingQuantity > availableInCurrentTier, the loop restarts using the same price as before because getTokenPrice checks using the tier price using the currentTier variable which doesn't get updated as the loop exits (simulatedTier is incremented but isn't used in the getTokenPrice check). This allows users to purchase up to the maximum allocation per user with the same price of tokens across different tiers, leading to economic loss for the protocol.

Recommendation

Update the currentTier variable within the while loop to prevent stale values from being accessed.

Status

Resolved

6. advanceTier() doesn’t update tierMinted when quantity of tokens to mint overflows into the next tier.

Description

_advanceTier() is called to increment the tier when all the nodes in the current tier have been sold. When this happens, the tierMinted is reset to 0 but this doesn’t account for when nodes in the new tier are bought. tierMinted should reflect the amount of nodes purchased in the new tier for proper accounting purposes else when users try to call buyNodes on the next call, the value returned will make the available nodes in the current tier always return a wrong value.

```
uint256 availableInCurrentTier = tierSizes[simulatedTier] - simulatedTierMinted;
```

Recommendation

After a successful simulation, the tierMinted variable should store the same value as remainingQuantity.

Status

Resolved

Medium Severity Issues

7. generateReferralCode not unique for addresses

Description

In solidity, all addresses are uint160 values, ranging from uint160(0) to type(uint160).max, therefore all addresses can be reduced to base 10 numbers.
In `generateReferralCode`, to get a ReferralCode for an address, a simple modulo is done. Modulo 1000000000.

The issue with using such a simple method is that multiple addresses when computed will give the same referral code. for example:
`0x0BA502c94dfDE079fC71f1234F4808a0d8396c02` and `0x602a41adc8968bBFfd3F97d310A9E6Da8e9e6c02` both have the same referral code of "2".
The root cause of this is the value 1000000000 is relatively small in solidity, therefore it cannot provide uniqueness.

This issue would actually prevent addresses from registering as referrals.

Recommendation

Take the ` keccak256()` hash of any address, that can then be converted to a string for uniqueness sake. Also an address itself is already unique.

Status

Resolved

8. All users can get discounts

Description

All users can get 25% of their purchases by simply calling the registerReferralCode with another address, and then use that as a code as their referral. This is because the referral system is open to all users at all times. The gas cost to register another address as a referral is already offset by the cut from the discount on their purchases.
20% referral bonus and 5% discount would also help previous buyers just resell their older nfts at a cheaper price than the mint.

Recommendation

It is advised that the referrer registration function be set to an admin function to actual set whatever offchain step necessary before any random address gets to be a referrer

Status

Resolved

Further Comment

Users can bypass the frontend to purchase the token through the contract. However since the only issue is discounted sales for users, then that is fine.

9. Stale Price from chainlink

Description

There are multiple problems when validating the price feed data that is used in `getTokenPrice()`.
- Price staleness not checked
- Sequencer downtime not checked

By not checking the price staleness or sequencer uptime, token amounts calculated in `getTokenPrice()` can be wrong in severe cases, allowing users to purchase tokens at discounted rates.

Recommendation

Chainlink recommends using their data feeds along with some controls to prevent mismatches with the retrieved data, Your application should track the latestTimestamp variable or use the updatedAt value from the latestRoundData() function to make sure that the latest answer is recent enough for your application to use it. If your application detects that the reported answer is not updated within the heartbeat or within time limits that you determine are acceptable for your application, pause operation or switch to an alternate operation mode while identifying the cause of the delay.Also to handle L2 Sequencer downtimes, use: https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code

Status

Acknowledged

Further Comment

The use of latestRoundData() does not always return the latest price generally. It only returns the last price updated by chainlink. This is not a common issue, but in cases of price crashes such as LUNA, this was an issue for protocols, See this post mortem of a similar hack in the past: **Blizz Finance, Venus Protocol - REKT**
In this case users will be able to use the undervalued token to buy multiple Nodes.

10. Referral rewards cannot be withdrawn

Description

The contract accrues referral rewards for those who register, values are stored on the contract, however there is no function that enables rewards of the actual accrued rewards.

Recommendation

Add withdrawal function for referral to withdraw their total accrued rewards.
This should be done after High 3 has been resolved

Status

Resolved

11. Withdraw and withdrawToken leaves no token left for the referrers

Description

The `withdraw()` and `withdrawToken()` function transfer the total ETH/Token balance to the admin address. This completely removes referrer rewards meaning referrers will not be able to withdraw their rewards if they have to do so using the `withdrawReferralReward()` function.
This issue will cause permanent damage for all referrers with ETH rewards and temporary DOS for users with token rewards `

Recommendation

- Add a mapping that tracks token address to total reward accrued. Subtract that value from the contract balance before any admin withdrawal.

Status

Resolved

Auditor's Remarks

Referrer's rewards are now transferred during the node buying process fixing the issue before it can arise.

12. TokenURI Reverts

Description

The latest update to the contract removes the tokenURI override which enables the use of the tokenURI directly with the baseImageURI() that was created on this contract.
SInce the full batch method from Thirdweb wasn't used, but instead the setBaseImageURI() was created, the tokenURI() needs to be overridden.

Recommendation

Create the tokenURI function in contract that will return the appropriate URI by utilizing the baseImageURI.

The first line of the function has to be:
function tokenURI(uint256 tokenId) public view virtual override returns (string memory)

From Second Review

The tokenURI() view function from thirdweb's library is explicitly described to be overriden to add your project's custom logic for describing your token URI but within the contracts this doesn't override the existing implementation. Everytime the tokenURI() function is called the function will revert, and no data will be returned, making the NFTs not have any data readable from them.

Recommendation

Properly implement the tokenURI() function.

Status

Resolved

13. referrerToReferees mapping not updated.

Description

The mapping is declared but not implemented within the current codebase, as even the _isReferee() function which depends on it is redundant. If there is meant to be some sort of limit or check for referrers and referees within the contracts that affect their logic, it is not implemented here as it should.

Recommendation

Contract storage space is costly and should be properly managed. As the contract is already well over the 24kb size limit, it is advisable to cut down on unused variables, mappings and functions to reduce the bytecode size and gas costs for user's interaction.

Status

Resolved

14. Buyer can now be the referrer

Description

When the referrer is the buyer, they can get even more discounts on their purchases than they normally should. Users can game the system to pay less for more tokens this way.

Recommendation

Include a check for the buyer not to be the referrer.

Status

Resolved

Low Severity Issues

15. EMIT EVENTS

Description

None of the functions emit events, both admin functions and user functions. Events are important for indexers monitoring contracts and providing offchain data.

Recommendation

Consider adding an event to:

- setBaseImageURI()
- setMaxPerWallet()
- setTotalSupply()
- setMaxTiers()
- registerReferralCode()
- buyNode()
- withdraw()
- withdrawToken()

Status

Resolved

16. Contract size exceeds 24kb limit

Description

The smart contract currently exceeds the 24kb limit (introduced in the Spurious Dragon fork) and stands the risk of the code not being deployed to the blockchain without the optimizer enabled.

Recommendation

Modularize the code, break it into smaller sections using libraries or proxy contracts, and enable the optimizer when deploying the smart contracts.

Status

Resolved

17. Dead Code

Description

The event and variable below are unused.
L34: mapping(address => mapping(address => uint256)) public referralRewards;
L56: event BaseImageURISet(string newBaseImageURI);
L214: function getReferralRewards(...)
L408: pricePerNodeInEth = tierPrices[currentTier];

Recommendation

Remove the unused code.

Status

Partially Resolved

Informational Issues

18. Use SafeTransfer

Description

Although tokens being used in this contract are known, it is best practice to use SafeERC20 to handle safe transfers of fungible tokens, this would prevent unforeseen scenarios

Recommendation

Use the SafeERC20 library from open zeppelin

Status

Acknowledged

Further Comment

Previous:
require(token.transferFrom(msg.sender, address(this), totalPrice), “Token transfer failed”);

Proper way:
CurrencyTransferLib.safeTransferERC20(paymentToken,msg.sender, address(this), totalPrice);

Kindly note the parmeter arrangement for each different transfer type.

19. Tier Prices not fully changed

Description

Tier prices were changed from Tier 1 to Tier 12 by a factor of 1000, However it wasn't updated from 13-77.

Recommendation

Update all prices

Status

Acknowledged

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

Some issues of High, medium and Low severity were found. Some suggestions, gas optimizations 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 Quranium. 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 Quranium. 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 Quranium 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

Quranium

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com