QuillAudits

Audit Report
November, 2023

For

ANRYTON

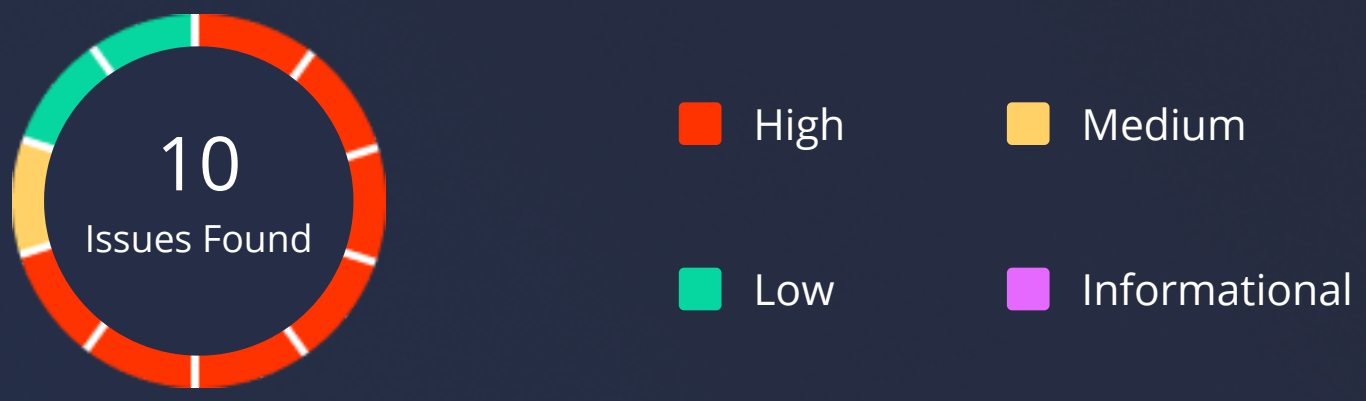
Table of Content

Table of Content

Executive Summary

Project Name	Anryton
Project URL	*https://anryton.com/home*
Overview	Anryton is a token contract with 400 million token supply. Sale contract is used to create different sales for the token. Stake contract is used for staking, vesting, depositing tokens to get rewards after staking for cliff periods of time which are linearly distributed over a period of months.
Audit Scope	*https://github.com/kuldeep349/anrytonContract*
Contracts in Scope	1) Aryton.sol 2) sales.sol 3) stake.sol
Commit Hash	ec82637f71d95a9ae854f3ed5aa0ed458f5f22ca
Language	Solidity
Blockchain	Polygon
Method	Manual Analysis, Functional Testing, Automated Review
Review 1	3rd November 2023 - 17th November 2023
Updated Code Received	21st November 2023
Review 2	22nd November 2023 - 24th November 2023
Fixed In	6ef89a569c409001d70fb30dc5f5733df78f82b3

Number of Security Issues per Severity



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	1	0	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	6	1	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
- 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 behaviour.
- Token distribution and calculations are as per the intended behaviour 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, Truffle, 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.

A. Contract - Stake

High Severity Issues

A.1: Wrong _maturedAmt calculation while un-staking and withdrawing tokens

Description

In Stake contract when user tries to un-stake the amount it is calculated over the period of 13 to 18 months after locking(stake for) of 12 months. But for the **unstakeTokens()** function when called for 13-17 months the amount released is correct but for the 18th month it does not calculate the values correctly.

Remediation

To resolve the issue please rewrite the logic correctly to let user un-stake tokens.

Status

Resolved

A.2: User can only unstake once

Description

From issue A1. we can say that if a user did un-stake tokens for 17 months in one call then wants to un-stake remaining tokens but he won't be able to due to "Arithmetic over/ underflow".

The line **stakings[_user][i].unStakedAmount += uint160(releaseAmount)** stores the both 1st withdrawal and 2 withdrawal values which makes it greater what actual amount is and then causes arithmetic over/underflow issue while subtracting.

Remediation

To resolve the issue please rewrite the logic correctly to support unstaking/withdrawal any number of times.

Status

Resolved

A.3: View functions vesting() and getMaturedStakeAmt() give different matured token amounts than the actual transferred amount in withdraw() and _unStakeTokens()

Description

vesting() and getMaturedStakeAmt() functions let users know how many tokens they are going to receive if they call withdraw() and _unStakeTokens() (via unStakeOrLock()), respectively. However, the amount returned by these view functions differs vastly from what is transferred by state-changing functions.

Remediation

To resolve the issue please rewrite the logic correctly.

Status

Resolved

A.4: withdraw() transfers same number of tokens after 40 seconds(>7*5) as transferred after 215 seconds(>42*5)

Description

The number of tokens transferred between the 7 to 42 months is the same as transferred after completion of 42 months tenure.

Remediation

To resolve the issue please rewrite the logic correctly.

Status

Resolved

A.5: Rounding error issue

Description

In contract stake, if user stakes or vests the tokens and after the cliff period tries to unstake/withdraw the tokens, then user will get less than what is expected because of the rounding error caused by getMaturedStakeAmt() and _vesting() functionality.
In getMaturedStakedAmt() function, line uint256 unStakePerMonth = userStaking.stakeAmount / 6

A.5: Rounding error issue

Description

Where the staked amount is divided by 6, causing rounding of the value
Also, in _vesting function in loops _amount+=_userDeposit.depositAmount / 12 which is either divided by 12 or 24 according to sale, causes a rounding issue.

For example, if the user stakes 100 tokens and after a cliff period of 6 months tries to unstake then 100/6 = 16.6666666, but because solidity does not support decimal values, it'll be rounded to 16 for 6 months, it'll be 16*6 = 96, causing loss of 4 tokens.

Remediation

Please make sure to fix the issue by multiplying , dividing it with 10**18.

Status

Acknowledged

A.6: Wrong age calculation in _calcMarketting()

Description

Every time _m is greater than 18 then age=1, the rest of the conditions are skipped.

Remediation

Conditions should be like below:

```
if (_m == 42) age = 5;
else if (_m > 36) age = 4;
else if (_m > 30) age = 3;
else if (_m > 24) age = 2;
else if (_m >18) age = 1;
```

Status

Resolved

Medium Severity Issues

A.7: String length comparison in _compareEqual() will lead to more number of tokens withdrawn than intended

Description

vesting() function compares the sale name for determining _amount, which is calculated twice if the sale name is either “ADVISORS” or “RESERVES”, which have equal lengths so both conditions become True.

1st Condition:
https://github.com/kuldeep349/anrytonContract/blob/ec82637f71d95a9ae854f3ed5aa0ed458f5f22ca/0x6CB0c296F81175DE85Ab2b24359E0D25195Bd317/Stake.sol#L513

2nd Condition:
https://github.com/kuldeep349/anrytonContract/blob/ec82637f71d95a9ae854f3ed5aa0ed458f5f22ca/0x6CB0c296F81175DE85Ab2b24359E0D25195Bd317/Stake.sol#L522

Remediation

Instead of comparing length, compare their keccak256 hashes.

```
function compareEqual(
                string memory a,
                string memory b,
        ) public pure returns(bool) {
                return (keccak256(bytes(a)) == keccak256(bytes(b)));
        }
```

Status

Resolved

Low Severity Issues

A.8: In _lockTokens() wrong address is being used for totalLocks

Description

totalLocks of _to address should be used as locking of _to address's token takes place.

Remediation

Instead of msg.sender use _to in this **line**.

Status

Resolved

A.9: getLeftBalance() returns the wrong balance

Description

Staked and locked tokens need not be subtracted from the token balance as the deposited balance does not have any relation with staked and locked tokens.

Remediation

Return the user's balance as it is without subtracting the staked and locked balance.

Status

Resolved

Informational Issues

No issues were found.

General Recommendations

- Too many integer casting in the contract which causes gas increase. If possible, make them like either one uint96, uint128, uint196.
- The deposit function has the wrong spelling make sure to correct it.

B. Contract - Anryton

High Severity Issues

B.1: Wrong total supply value

Description

In the token contract Anryton according to the whitepaper the total supply is 400 million, but in the contract only 75 million is mentioned. Also, the variable is constant so it won't be possible to change or mint more tokens as the mint function checks if the tokens minted are more than 75 million.

Remediation

Make sure to fix the token supply value and change the sale values according to it.

Status

Resolved

Anryton Team's Comment

As there are different sales to be held they'll manage the capped amount for each sale manually.

Medium Severity Issues

No issues were found.

Low Severity Issues

No issues were found.

Informational Issues

No issues were found.

Functional Tests

Tests Performed:

https://gist.github.com/aga7hokakological/5911bd9acb97012b5f7f3ef4571650f7

Some of the tests performed are mentioned below:

- Number of tokens transferred after anod in between the completion of tenure of staking
- Number of tokens transferred after and in between the completion of tenure of vesting
- Do the results of view functions and storage functions of vesting and staking match with their respective state-changing functions
- Can the user un-stake multiple times before the completion of tenure
- Is the user getting the intended amount after staking, locking, and vesting
- The test should fail if user tries to withdraw before un-stake period or cliff time

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

Some issues of Hogh, 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 Anryton 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 Anryton 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 Anryton to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

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.

850+
Audits Completed

$30B
Secured

$30B
Lines of Code Audited

Follow Our Journey

Audit Report
November, 2023

For

ANRYTON

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com