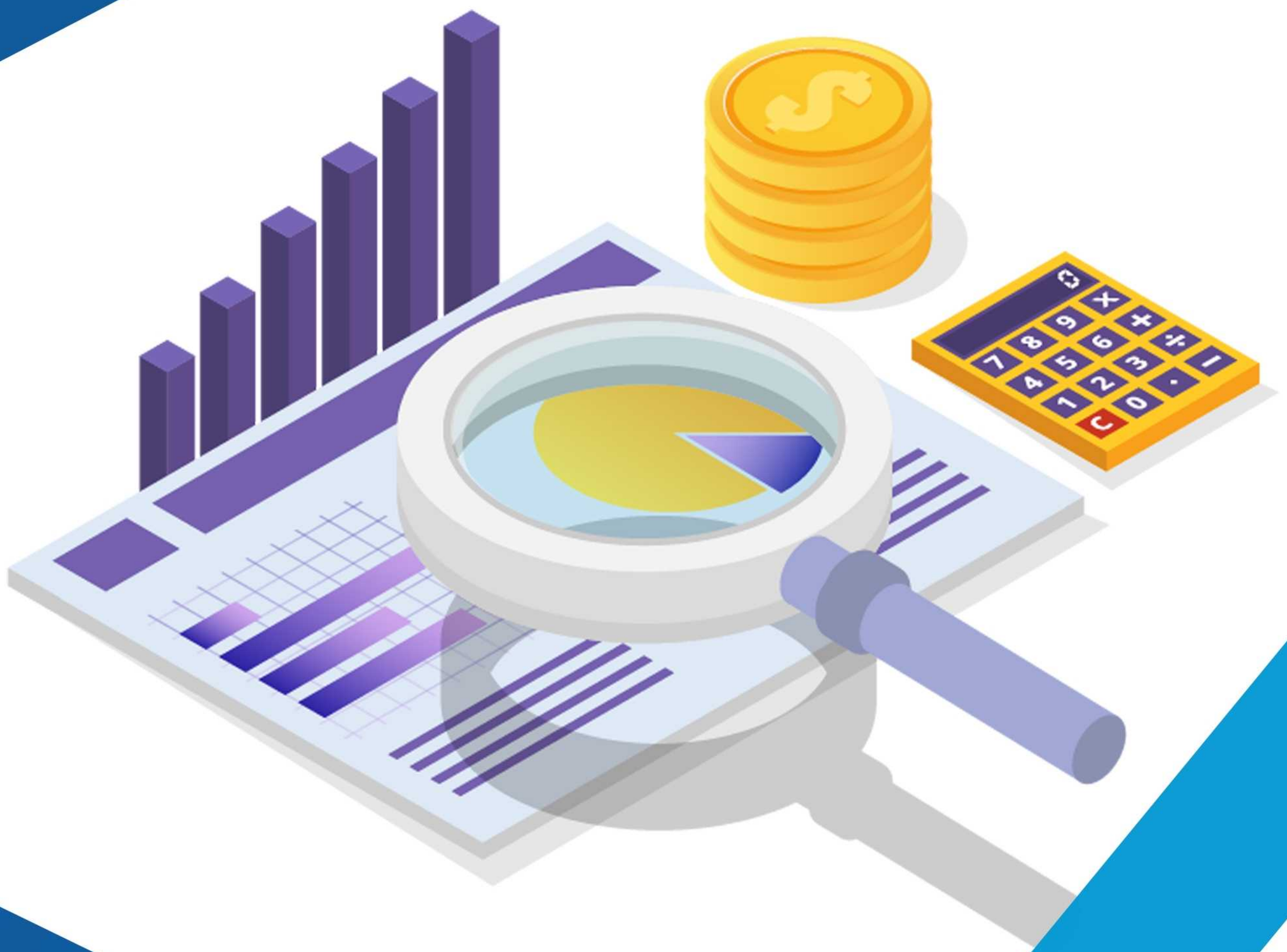
QuillAudits



Audit Report
June, 2021

Contents

Audit Details and Target	01
Scope of Audit	04
Techniques and Methods	05
Issue Categories	06
Issues Found – Code Review/Manual Testing	07
Summary	10
Disclaimer	11

Audit Details and Target

1. Contract

https://github.com/bytenext/avatar-art-staking/tree/main/contracts
Commit Hash: bf9d75dc7841ff320a6a29c927bd955e2585a927

2. Audit Target

- To find the security bugs and issues regarding security, potential risks, and critical bugs.
- Check gas optimisation and check gas consumption.
- Check function reusability and code optimisation.
- Test the limit for token transfer and check the accuracy in decimals.
- Check the functions and naming conventions.
- Check the code for proper checks before every function call.
- Event trigger checks for security and logs.
- Checks for constant and function visibility and dependencies.
- Validate the standard functions and checks.
- Check the business logic and correct implementation.
- Automated script testing for multiple use cases including transfers, including values and multi transfer check.
- Automated script testing to check the validations and limit tests before every function call.
- Check the use of data type and storage optimisation.
- Calculation checks for different use cases based on the transaction, calculations and reflected values.

Functions list and audit details

Major Functions

- createLockStage()
- getAnnualProfit()
- getBnuTokenAddress()
- getLockStages()
- getStopTime()
- getTotalStaked()
- getUserEarnedAmount()
- getStakingUsers()
- getStakingHistories()
- getUserLastEarnedTime()
- getUserLastStakingTime()
- getUserRewardPendingTime()
- getUserStakedAmount()
- getWithdrawalHistories()
- setLockStage()
- setLockStageActiveStatus()
- setAnnualProfit()
- stake()
- stop()
- withdraw()
- _calculateInterest()
- _calculatePendingEarned()
- _isUserStaked()
- _getUserLastEarnedTime()
- _getUserLastStakingTime()
- _getUserRewardPendingTime()

Overview of The Contract

Avatar ArtStaking Contract is a staking platform for the tokens. Code is written in solidity and used to perform staking logic. Users stake a specific token amount for a specific lock stage and withdraw the staked amount in a specific time stage. The contract is owned by the contract owner and can be managed and owned by the contract owner for the managing function of the contract, here we can change the owner as per the logic and requirements.
Staking is based on the time and rewards that can be managed as per the logic of the contract. Users can check the active state of the staking and rewards or profit.
Tokens are safe in this contract and pass all the major security checks of the auditing and testing process.

Tokenomics

As per the information provided, the tokens generated will be initially transferred to the contract owner and then further division will be done based on the business logic of the application. Tokens cannot be directly purchased from the smart contract, so there will be an additional or platform that will help users to purchase the tokens. Tokens can be held, transferred and delegated freely. Tokens are generated based on the fixed flow in the supply, which can be checked by total supply. Staking contract is used to stake tokens and earn interest with time.

Scope of Audit

The scope of this audit was to analyse AvatarArtStaking smart contracts codebase for quality, security, and correctness.

Checked Vulnerabilities

The smart contract is scanned and checked for multiple types of possible bugs and issues. This mainly focuses on issues regarding security, attacks, mathematical errors, logical and business logic issues. Here are some of the commonly known vulnerabilities that are considered:

- TimeStamp dependencies.
- Variable and overflow
- Calculations and checks
- SHA values checks
- Vulnerabilities check for use case
- Standard function checks
- Checks for functions and required checks
- Gas optimisations and utilisation
- Check for token values after transfer
- Proper value updates for decimals
- Array checks
- Safemath checks
- Variable visibility and checks
- Error handling and crash issues
- Code length and function failure check
- Check for negative cases
- D-DOS attacks
- Comments and relevance
- Address hardcoded
- Modifiers check
- Library function use check
- Throw and inline assembly functions
- Locking and unlocking (if any)
- Ownable functions and transfer ownership
- checksArray and integer overflow possibility checks
- Revert or Rollback transactions check

Techniques and Methods

- Manual testing for each and every test cases for all functions.
- Running the functions, getting the outputs and verifying manually for multiple test cases.
- Automated script to check the values and functions based on automated test cases written in JS frameworks.
- Checking standard function and check compatibility with multiple wallets and platforms
- Checks with negative and positive test cases.
- Checks for multiple transactions at the same time and checks d-dos attacks.
- Validating multiple addresses for transactions and validating the results in managed excel.
- Get the details of failed and success cases and compare them with the expected output.
- Verifying gas usage and consumption and comparing with other standard token platforms and optimizing the results.
- Validate the transactions before sending and test the possibilities of attacks.

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.
SmartCheck.

Code Review / Manual Analysis
Manual Analysis or review of code was done to identify new vulnerability 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 automated analysis were manually verified.

Static Analysis
Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step a series of automated tools are used to test security of smart contracts.

Gas Consumption
In this step we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

High severity issues

Issues that must be fixed before deployment else they can create major issues.

Medium level severity issues

These issues will not create major issues in working but affect the performance of the smart contract.

Low level severity issues

These issues are more suggestions that should be implemented to refine the code in terms of gas, fees, speed and code accuracy

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.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	0	0
Closed	0	0	2	1

Issues Found – Code Review / Manual Testing

High severity issues

No issues found in this category

Medium severity issues

No issues found in this category

Low level severity issues

1.

```
221     /**
222      * @dev See IAvatarArtStaking
223      */
224     function stake(uint lockStageIndex, uint amount) external override isRunning returns(bool){
225         //CHECK REQUIREMENTS
226         require(lockStageIndex < _lockStages.length, "Out of length");
227         require(amount > 0, "Amount should be greater than zero");
228 
229         //Check for minimum staking amount
230         LockStage memory lockStage = _lockStages[lockStageIndex];
231         require(lockStage.isActive, "This staking stage is inactive");
232         if(lockStage.minAmount > 0){
```

Gas estimation failed

Gas estimation errored with the following message (see below). The transaction execution will likely fail. Do you want to force sending?
Internal JSON-RPC error. { "code": 3, "message": "execution reverted: Out of length", "data":
"0x08c379a0000000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000000000000000d4f7574206f66206c656e677468000000000000000000000000000000000000" }

Contract: AvatarArtStaking.sol
Line: 224
Issues: Higher gas usage
Reasons: Function has many internal events that use more gas; this should be minimised if possible.

Status: Fixed and Closed

2.

```
276     /**
277      * @dev See IAvatarArtStaking
278      */
279     function withdraw(uint lockStageIndex, uint amount) external override returns(bool){
280         require(lockStageIndex < _lockStages.length, "Out of length");
281         //Calculate interest and store with extra interest
282         _calculateInterest(_msgSender());
283 
284         IERC20 bnuTokenContract = IERC20(_bnuTokenAddress);
```

Contract: AvatarArtStaking.sol
Line: 279
Issues: Failing on the test case “Revert initially with a message of condition fail.”
Reasons: Modifier needed to check the amount at the initial stage of calling withdraw function for the incorrect amount. Ex. 0 in amount may fail with gas loss.

Status: Fixed and Closed

Informational

3. Commenting can be improved for a better understanding of the functions for those users who wants to use smart contract functions directly and check the logic.

Status: Fixed and Closed

Functional Test Table

Function Names	Technical results	Logical results	Overall
createLockStage	Pass	Pass	Pass
getAnnualProfit	Pass	Pass	Pass
getBnuTokenAddress	Pass	Pass	Pass
getLockStages	Pass	Pass	Pass
getStopTime	Pass	Pass	Pass
getTotalStaked	Pass	Pass	Pass
getUserEarnedAmount	Pass	Pass	Pass
getStakingUsers	Pass	Pass	Pass
getStakingHistories	Pass	Pass	Pass
getUserLastEarnedTime	Pass	Pass	Pass
getUserLastStakingTime	Pass	Pass	Pass
getUserRewardPendingTime	Pass	Pass	Pass
setLockStage	Pass	Pass	Pass
setLockStageActiveStatus	Pass	Pass	Pass
setAnnualProfit	Pass	Pass	Pass
stake	Fixed and Pass	Pass	Fixed and Pass
stop	Pass	Pass	Pass
withdraw	Fixed and Pass	Pass	Fixed and Pass

Closing Summary

All the issues are fixed and changed as per the suggestion. The contract is now working fine for the staking, withdrawal and other business logic of the contract.
The contract is audited and passed for the logical and technical parameters. Security of tokens are checked and commenting is improved for better understanding for the users. Staking and time logic is also checked for multiple test cases and scenarios.

Disclaimer

QuillHash audit is not a security warranty, investment advice, or an endorsement of the AvatarArtStaking platform. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the AvatarArtStaking Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

audits@quillhash.com