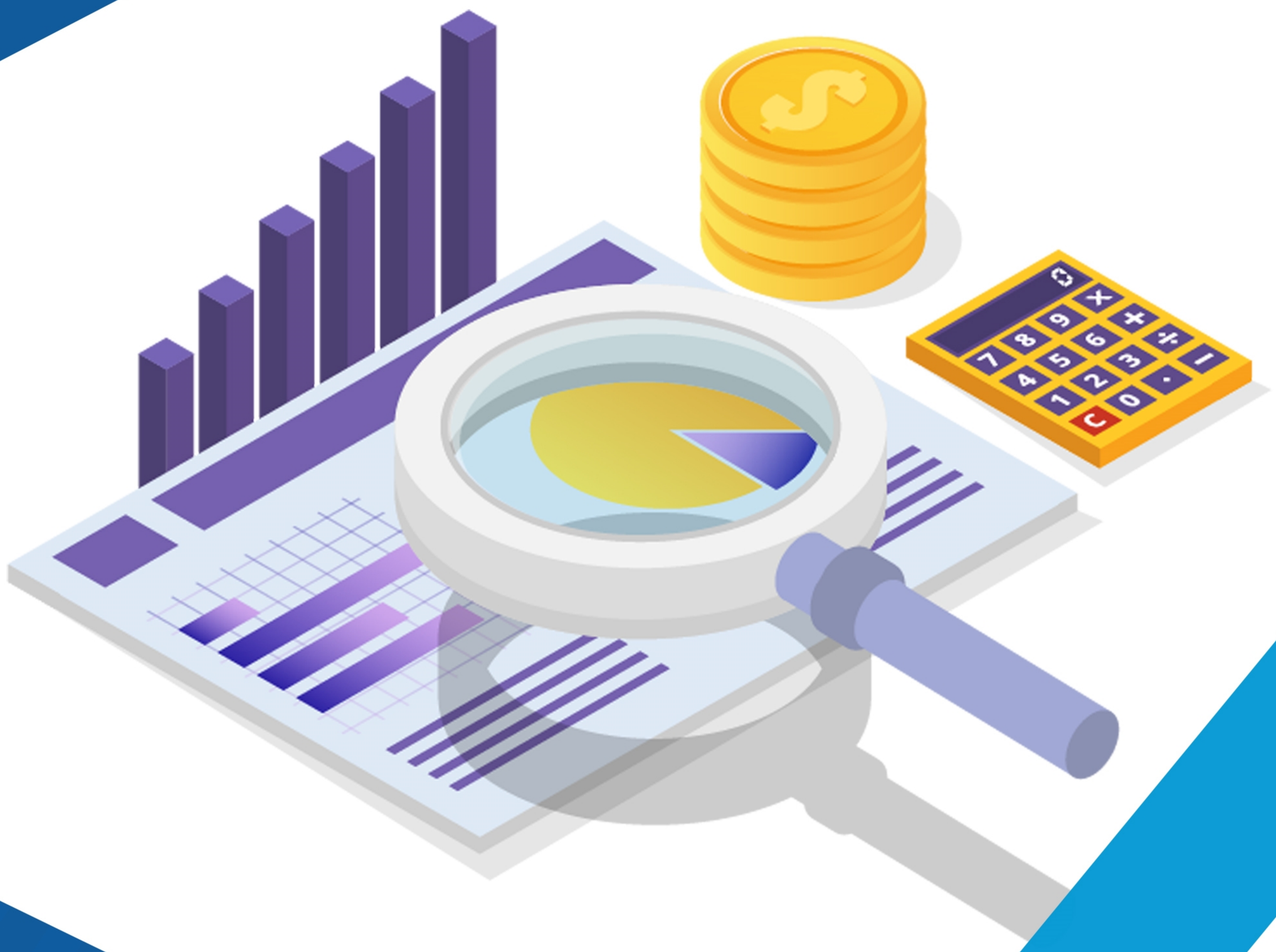
QuillAudits



Audit Report
April, 2021

Contents

Scope of Audit

The scope of this audit was to analyze and document Dart smart contract codebase for quality, security, and correctness.

Check 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
- Malicious libraries
- 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 behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token 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.
SmartCheck.

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.

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.

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

Every issue in this report has been assigned with 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 to be fixed before moving to a live environment.

Medium level 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 level severity issues

Low level severity issues can cause minor impact and or 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 severity four issues which 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	1	0	0
Closed	3	2	8	0

Introduction

During the period of **April 8th, 2021 to April 11th, 2021** - Quillhash Team performed a security audit for **Dart** smart contracts.
The code for the audit was taken from following the official Github link:

https://github.com/artl-dev/dART-contracts/tree/master/contracts

Commit Hash - **bdc8babc0a63e359aa9eac9975294255a026aef1**

A. Contract Dart - DartVesting

Issues Found – Code Review / Manual Testing

High severity issues

1. **addRecipients** function fails to provide adequate logic for similar **recipient addresses** passed in arguments more than once.
   Line no - 63-93

Status: Closed

Description:
As per the current design, the **addRecipients** function allows the owner to pass the recipients addresses as an array and assign the allocations to those addresses.

However, if a recipient address is passed more than one time as an argument to the **addRecipients** function, then the function doesn't handle the scenario very effectively. As of now, the function, in the above-mentioned scenario, simply overrides the **totalAmount** of the recipient with the new **allocation** passed and doesn't take into account the existing/initial **totalAmount** assigned to that recipient address.

For instance, if a particular recipient address **0xabcxyz** was passed initially with a total allocation of 5000 tokens. And the similar address(**0xabcxyz**) is passed again with an allocation of just 100 tokens, then the totalAmount stored in the contract for the recipient address(**0xabcxyz**) is only **100 tokens** instead of **5100 tokens**.

```
function addRecipients(address[] memory _newRecipients, uint256[] memory _allocatio
    public
    onlyOwner
{
    // Only allow to add recipient before the counting starts
    require(!isStartTimeSet || startTime > block.timestamp);
    require(_newRecipients.length == _allocations.length);

    for (uint256 i = 0; i < _allocations.length; i ++) {
        require(_newRecipients[i] != address(0));

        unallocatedAmount = unallocatedAmount.add(
            recipients[_newRecipients[i]].totalAmount
        );

        require(_allocations[i] > 0 && _allocations[i] <= unallocatedAmount);

        if (recipients[_newRecipients[i]].totalAmount == 0) {
            recipientAddresses.push(_newRecipients[i]);
        }

        recipients[_newRecipients[i]] = VestingSchedule({
            totalAmount: _allocations[i],
            amountWithdrawn: 0
        });
```

This will lead to a completely unwanted scenario as the allocation of tokens to the respective recipient addresses will be disturbed if a similar address is passed more than one time.

Recommendation:
The above-mentioned scenario must be handled adequately within the addRecipients function. As of now, the issue can be effectively handled in 2 ways:

- If the repetition of any recipient address is not supposed to take place, the addRecipients function must include a procedure to check that the recipient addresses passed as arguments to this function are not available in the contract beforehand. Thus, ensuring that a particular address is passed only once.

- If the recipient addresses might be repeated while passing as an argument to the addRecipient function, then the previous token allocations of the repeated recipient addresses must be taken into consideration while adding the new allocations.

2. **UNLOCK_TGE_PERCENT** is wrongly assigned in the TeamVesting.sol contract.
   Contract Name - **TeamVesting**
   Line no - 10

   Status: Closed

Description:
As per the DART tokenomics, the **Unlock TGE Percent** for the team should ZERO.
However, in the TeamVesting contract, the state variable **UNLOCK_TGE_PERCENT** has been assigned a value of 15%.
This will lead to an unexpected scenario where the tokenomics implemented in the vesting contract doesn't follow the actual tokenomics design.

Recommendation:
The Vesting contracts should match the actual Tokenomics.
In case the tokenomics has been changed, the documentation must be updated with the latest token distribution and vesting rules.

Medium severity issues

3. Loops are extremely costly

Line no - 10
Note: The issue is still open at Line 69

Status: Partially Closed

Description:
The **for loops** in the **DartVesting** contract include state variables like **.length** of a non-memory array in the condition of the for loops.
As a result, these state variables consume a lot more extra gas for every iteration of the loop.

The following functions include such loops at the mentioned lines:

- **addRecipients function** at Line 69

Recommendation:
It’s quite effective to use a local variable instead of a state variable like .length in a loop.

For instance,

```
local_variable = recipientAddresses.length
 for (uint i = 0; i < local_variable; i ++) {
   address recipientAddr = recipientAddresses[i];
          if (recipients[recipientAddr].totalAmount > 0) {
              uint256 amount =
recipients[recipientAddr].totalAmount.mul(unlockTGEPercent).div(100);
              recipients[recipientAddr].totalAmount =
recipients[recipientAddr].totalAmount.sub(amount);
              dARTToken.transfer(recipientAddr, amount);

   }
```

4. Multiplication is being performed on the result of Division

Line no- 132-134

Status: Closed

Description:
During the automated testing of the DartVesting.sol contract, it was found that the **vested** function in the contract is performing multiplication on the result of a Division.

```
131
132         uint256 releaseAmountPerPeriod = _vestingSchedule.totalAmount.div(releasePeriods);
133
134         uint256 vestedAmount = period.sub(lockPeriods).mul(releaseAmountPerPeriod);
135         return vestedAmount;
136     }
```

Integer Divisions in Solidity might truncate. Moreover, this performing division before multiplication might lead to loss of precision.

Recommendation:
Solidity doesn’t encourage arithmetic operations that involve division before multiplication. Therefore the above-mentioned function should be checked once and redesigned if they do not lead to expected results.

5. **withdraw function should include require statement instead of IF-Else Statement**
   Line no - 146-158

   Status: Closed

   Explanation:
   The **withdraw** function includes an **if statement** at the very beginning of the function to check if the total amount of tokens of the caller of the function is equal to ZERO.

```
146     function withdraw() public {
147         VestingSchedule storage vestingSchedule = recipients[_msgSender()];
148         if (vestingSchedule.totalAmount == 0) return;
149
```

   However, in order to be able to access the withdraw function the caller must have the **totalAmount** greater than ZERO. This is a strict check as the function should revert back if the total amount of tokens is ZERO.

   Therefore, in order to check for such validations in a function, **require statements** are more preferable and effective Solidity rather than **IF-Else statements**. While it helps in gas optimizations, it also enhances the readability of the code.

   Recommendation:
   Use **require statement** instead of **IF statement** in the above-mentioned function line.
   For instance,
   require(vestingSchedule.totalAmount > 0,”Error MSG: Total Amount is ZERO”);

Low level severity issues

6. **Redundant State Variable Update**
   Line no - 60

   Status: Closed

Description:
The boolean state variable **isStartTimeSet** is being set to false within the constructor.
However, the default value of a boolean state variable is already **FALSE** in Solidity. Therefore, it doesn't require an explicit updation in the constructor.

Recommendation:
State Variables updates should not be redundant.

7. External visibility should be preferred

Status: Closed

Description:
Those functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
This will effectively result in Gas Optimization as well.
Therefore, the following function must be marked as **external** within the contract:

- **addRecipients**
- **setStartTime**
- **withdraw**
- **withdrawable**

Recommendation:
The above-mentioned functions should be assigned external visibility.

8. Absence of Error messages in Require Statements

Status: Closed

Description:
None of the **require statements** in the contract doesn't include an error message. While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

Recommendation:
Error messages should be included in every require statement

9. NatSpec Annotations must be included

Status: Not Considered

Description:
Smart Contract does not include the NatSpec Annotations adequately.

Recommendation:
Cover by NatSpec all Contract methods.

B. Contract Dart - DartToken

High severity issues

1. Token Allocated in Contract does not match with actual Tokenomics Design

Line no - 24-37

Status: Closed

Description:

Some of the DART token allocation amounts for specific categories don’t match with the actual tokenomics document provided.

Mentioned below are some of the State Variables with different token Allocations than the actual tokenomics:

- Staking_Rewards
- Liquidity_Pool
- Marketing
- Treasury

Tokenomics in the Document Provided:

	Tokens
Seed round	3,000,000
Private round	15,000,000
IDO	3,890,000
Early supporters	200,000
Staking incentives	35,000,000
Liquidity provision	15,000,000
Marketing / Listings	20,000,000
Treasury	28,000,000
Team	17,000,000
Advisors	5,000,000

Tokenomics in the Document Provided:

```
27      uint256 public constant STAKING_REWARDS = 24500000 * (10**uint256(DECIMALS));
28      uint256 public constant LIQUIDITY_POOL = 9000000 * (10**uint256(DECIMALS)); /
29      uint256 public constant MARKETING = 14000000 * (10**uint256(DECIMALS)); // 9.
30      uint256 public constant TREASURY = 19600000 * (10**uint256(DECIMALS)); // 13.
31
```

Recommendation:

The Token Allocation in the Smart Contracts should match the actual Tokenomics.
In case the tokenomics has been changed, the documentation must be updated with the latest token distribution and vesting rules.

Low level severity issues

2. External visibility should be preferred

Status: Closed

Description:
Those functions that are never called throughout the contract should be marked as external visibility instead of public visibility.
This will effectively result in Gas Optimization as well.
Therefore, the following function must be marked as external within the contract:

- **setDistributionTeamsAddresses** at Line 56
- **distributeTokens** at Line 81

Recommendation:
The above-mentioned functions should be assigned external visibility.

3. User ETHER Units instead of Large Digits
Line no: 24-37

Status: Not Considered

Description:
Since the decimal points used for the DART token is **18**, the globally available Ether Units can be used instead of multiplying token allocation amounts with 10^18 while assigning them to State Variables. This will enhance the readability of the contract code.

Solidity provides some globally available units like **ether** which symbolizes **10^18**.
For instance,
uint256 public constant SEED_INVESTMENT = 3000000 ether;

Recommendation:
Consider using **Ether Units(ether)** instead of multiplying token allocation amounts with **(10**uint256(DECIMALS)**

4. **Absence of Error messages in Require Statements**

Status: Closed

Description:
None of the **require statements** in the contract doesn't include an error message. While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

Recommendation:
Error messages should be included in every require statement.

5. **NatSpec Annotations must be included**

Status: Not Considered

Description:
Smart Contract does not include the NatSpec Annotations adequately.

Recommendation:
Cover by NatSpec all Contract methods.

Closing Summary

Overall, smart contracts are very well written and adhere to guidelines.
All the high, medium and low severity issues found during the initial audit procedure have now been fixed.
No instances of Backdoor entry were found during the audit and the contract behaves as expected.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the **Dart 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 Dart 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

hello@quillhash.com