QuillAudits

AUDIT REPORT

October , 2024

For

2D3T

Table of Content

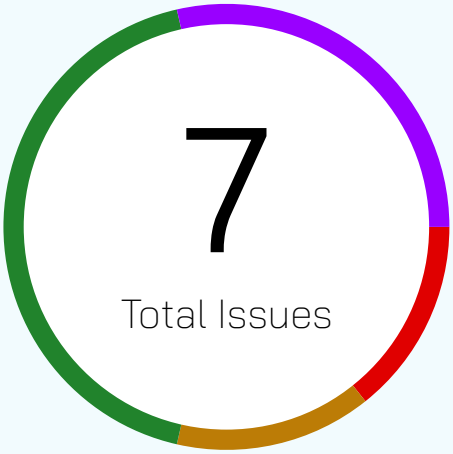
Executive Summary

Project name	2D3T
Project URL	https://2d3t-products.com/
Audit Scope	The Scope of the Audit is to analyse the Security, correctness and code quality of 2DT3 Codebase
Contracts in Scope	1. 2D3T.sol 2. Vesting.sol https://drive.google.com/drive/folders/19Sgp86Kp-taYjdeRFPJ44Bbrlemn2CCAC?usp=sharing
Commit Hash	NA
Language	Solidity
Blockchain	Polygon
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	26 September 2024 - 12 October 2024
Review 2	25th october 2024 - 28th october 2024
Fixed In	https://gitlab.com/infincube/crypto/ext_contracts/roof/smart_contracts/-/commit/0d7e870e0842626ae73f3744675583c721047099
2D3T Token Mainnet Address	https://polygonscan.com/address/0x61be2Fca17D3A6a393E64CCc3a6C61833d65C7D5#code

Note

QuillAudits Team confirm that the deployed code of the 2D3T token matches the version we audited. The only change identified is the token mint amount, which is set to 5,000 in the deployed contract, compared to 1 billion in the audited version.

Number of Issues per Severity



High	1 (14.29%)
Medium	1 (14.29%)
Low	3 (42.86%)
Informational	2 (28.57%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	0	0	0	0
Acknowledged	**1**	**1**	**3**	**2**
Partially Resolved	0	0	0	0

Checked Vulnerabilities

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

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] 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 statistical 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 Issues

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

Security vulnerabilities identified that must be resolved and are currently unresolved.

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

Centralization Risk - Owner can claim all vested tokens of the users

Acknowledged

Path
Vesting.sol

Function Name
emergencyWithdraw

Description
emergencyWithdraw function allows the contract owner to withdraw vested tokens from vesting positions despite the position not expiring.
If owner of contract becomes malicious, they can claim all the vested tokens and leaving the user with zero vesting tokens to claim.

```
function emergencyWithdraw(
    address _address,
    uint256[] calldata _vestingPositionIds
) external nonReentrant onlyOwner {
    uint256 amountToSend;
    uint256 vestingPositionId;
    uint256 amountChunk;

    // @audit _address is not checked for zero address
  for (uint256 i; i < _vestingPositionIds.length; ) {
        vestingPositionId = _vestingPositionIds[i];
        VestingPosition storage position = vestingPosition[_address][
            vestingPositionId
        ];

        amountChunk = position.amount;
        amountToSend += amountChunk;
        position.amount = 0;
```

```
        emit EmergencyWithdraw(
            _address,
            tokenHolderAccount,
            vestingPositionId,
            amountChunk
        );

        unchecked {
            ++i;
        }
    }

    vestingToken.safeTransfer(tokenHolderAccount, amountToSend); // @audit owner
can claim all of the tokens
}
```

This happens because the tokens are transferred to the tokenHolderAccount address (set during the initialization which can be owner addres as well)

Below is the hardhat test for the same

```
it.only("Owner claims all vested tokens", async function () {
    const vestingBalanceBefore = await vestingToken.balanceOf(vesting.address);

    expect(vestingBalanceBefore).to.be.equal(0);

    const depositAmount1 = WeiPerEther.mul(100_000);
    const depositAmount2 = WeiPerEther.mul(200_000);
    const depositAmount3 = WeiPerEther.mul(300_000);
    await vestingToken
        .connect(owner)
        .transfer(tokenHolderAccount.address, depositAmount1);
    await vestingToken
        .connect(owner)
        .transfer(tokenHolderAccount.address, depositAmount2);
    await vestingToken
        .connect(owner)
        .transfer(tokenHolderAccount.address, depositAmount3);
```

```
const tx1 = await vesting
    .connect(owner)
    .depositForUser(user.address, depositAmount1);
 const events1 = (await tx1.wait()).events;
 const depositedEvent1 = events1?.pop();
 const vestingPositionId1 = depositedEvent1?.args?.vestingPositionId;
 const tx2 = await vesting
    .connect(owner)
    .depositForUser(user.address, depositAmount2);
 const events2 = (await tx2.wait()).events;
 const depositedEvent2 = events2?.pop();
 const vestingPositionId2 = depositedEvent2?.args?.vestingPositionId;

 const tx3 = await vesting
    .connect(owner)
    .depositForUser(user.address, depositAmount3);
 const events3 = (await tx3.wait()).events;
 const depositedEvent3 = events3?.pop();
 const vestingPositionId3 = depositedEvent3?.args?.vestingPositionId;

 const vestedAmount = await vesting
    .connect(owner)
    .getTotalVestedAmount(user.address);
 const expectedVestedAmount = depositAmount1
    .add(depositAmount2)
    .add(depositAmount3);
 expect(await vestingToken.balanceOf(vesting.address)).to.be.equal(
    expectedVestedAmount
 );
 expect(vestedAmount).to.be.equal(
    expectedVestedAmount
 );
 // owner is performing emergency withdraw
 const tx = await vesting
    .connect(owner)
    .emergencyWithdraw(user.address, [
        vestingPositionId1,
        vestingPositionId2,
        vestingPositionId3,
    ]);
```

```
    const events = (await tx.wait()).events || [];
    for (let I = 0; I < events.length - 1; I++) {
        const event = events[I];
        const receiver = event?.args?.receiver;
        expect(await vesting.tokenHolderAccount()).to.be.equal(receiver);
    }

    const tokenHolderAccountBalance3 = await vestingToken.balanceOf(
        tokenHolderAccount.address
    );

    // tokenHolderAccount/owner has all the vested tokens
    expect(tokenHolderAccountBalance3).to.be.equal(
        tokenHolderAccountMintedAmount.add(expectedVestedAmount)
    );

    await mineBlocks(426, DAY); // after 426 days of vesting period

    // user wants to claim the vested tokens for position 3
    await vesting.connect(user).claim(vestingPositionId3);

    const userBalanceAfterClaim = await vestingToken.balanceOf(
        user.address
    );
    // balance is zero even though the claim call was successful
    expect(userBalanceAfterClaim).to.be.not.equal(depositAmount3);
    expect(userBalanceAfterClaim).to.be.equal(0);
});
```

Recommendation

Consider to remove the emergencyWithdraw function which adds centralization or transfer the tokens to the users itself during the emergency.

Comments from 2D3T Team

Acknowledged by the 2D3T team, no changes were made. 2D3T team mentioned that “Desired behaviour”.

Medium Severity Issues

Check-Effects-Interaction pattern is not followed

Acknowledged

Path

Vesting.sol

Function

claim and claimForUser

Description

claim and claimForUser allows user and contract owner for user to claim vestingToken tokens from the vesting contract after the pre-defined vesting period.
Both of these functions update the position's amount to zero after the transfer of vesting tokens which is not following the Check-Effects-Interaction pattern, external call is made before writing the state variables. Due to this there is possiblity of reentrancy.

```
function claim(uint256 _vestingPositionId) public nonReentrant {
...
// @audit check-effects-interaction pattern not followed
 vestingToken.safeTransfer(msg.sender, position.amount);

 emit Claimed(msg.sender, _vestingPositionId, position.amount);

 position.amount = 0; // state change
}
```

In case of claimForUser function, it doesn't have nonReentrant modifier as well.

```
function claimForUser(
 address _address
 uint256 _vestingPositionId
) external onlyOwner { // @audit not using nonReentrant modifier
...
// @audit check-effects-interaction pattern not followed
 vestingToken.safeTransfer(_address, position.amount);

 emit Claimed(_address, _vestingPositionId, position.amount);

 position.amount = 0; // state change
}
```

Recommendation

This is set to Medium severity because safeTransfer of ERC20 tokens won't involve the fallback/receive functions to reenter the function again through a contract.
Consider to follow Check-Effects-Interaction pattern in all functions and add nonReentrant modifier.

Low Severity Issues

Position's amount is not validated for zero amount

Acknowledged

Path

-

Function

-

Description

claim function in the contract allows users to claim vestingToken tokens from the vesting contract after the pre-defined vesting period.
This function is missing a check to verify that position's amount is not zero before the transfer like it is done in claimForUser function.

```
function claim(uint256 _vestingPositionId) public nonReentrant {
...

 VestingPosition storage position = vestingPosition[msg.sender][
_vestingPositionId
];

// @audit position's amount is not checked for zero amount
 vestingToken.safeTransfer(msg.sender, position.amount);
...
}
```

Recommendation

Consider to add a check the position's amount is not zero before the transfer of tokens.

Missing validation of _address parameter

Acknowledged

Path

-

Function

-

Description

emergencyWithdraw function in the contract is used to Withdraw positions to tokenHolderAccount without checking if finished in EMERGENCY ONLY cases. This function is missing zero address check on input _address like other functions do.

```
function emergencyWithdraw(
 address _address,
 uint256[] calldata _vestingPositionIds
) external nonReentrant onlyOwner {
// @audit _address is not checked for zero address
 uint256 amountToSend;
 uint256 vestingPositionId;
 uint256 amountChunk;
...
}
```

Recommendation

Consider to add a zero address check for the _address.

Missing event emission for conversion rate and decimals

Acknowledged

Path

-

Function

-

Description

During initialization of contract conversionRate and conversionRateDecimals variables are set which is a state change and there is no event emitted (ConversionRateChanged) for it.

```
function initialize(
 address _vestingTokenAddress,
 address _paymentTokenAddress,
 address _businessAccount,
 address _tokenHolderAccount,
 address _owner
) public initializer {
...
 conversionRate = 1_000000;
 conversionRateDecimals = 6;
...
 emit BusinessAccountAddressChanged(businessAccount);
 emit TokenHolderAccountAddressChanged(tokenHolderAccount);
// @audit missing event emission for conversion rate and decimals
}
```

Due to this sometimes subgraph/frontend will reflect wrong values.

Recommendation

Consider to emit the ConversionRateChanged event with the input values accordingly.

Informational Severity Issues

Mismatching contract name and file name

Acknowledged

Path

-

Function

-

Description

Based on the Solidity official docs Contract and library names should also match their filenames. Vesting is the filename and _2D3TVesting is the contract name which are different.

Recommendation

Consider to match file name and contract name.

Use Style Guide to structure the contracts

Acknowledged

Path

-

Function

-

Description

Solidity official style guide explains about standard order of layout. Also functions order can be changed based on this. Structure of referred contracts is difficult to read.

Recommendation

Consider to use order of layout documentation to change the structure of the contract.

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 2D3T. We performed our audit according to the procedure described above.

Some issues of High, low, medium and informational severity were found. Some suggestions, gas optimizations and best practices are also provided in order to improve the code quality and security posture.2D3T team acknowledged them all

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

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

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

October , 2024

For

2D3T

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com