# DoubleDice Audit

## Token and vesting

October 2021

By CoinFabrik

# Introduction

CoinFabrik was asked to audit the contracts for the DoubleDice project. First we will provide a summary of our discoveries and then we will show the details of our findings.

## Summary

The contracts audited are from the DoubleDice repository at https://github.com/DoubleDice-com/doubledice-token, but the first 3 iterations were made in another private git repository.

The first iteration of this audit is based on commit `6df7744c5727e666607839f9e3987174f18f2436`.

In the second iteration, developers fixed issues and did some new developments, including a complete redesign of the DoubleDice token and several name changes documented in the Names changed in the second iteration section. This iteration corresponds with commit `e6c997f409c47c79d61eb9db1fbfcc7958c60397`.

For the third iteration, in commit dd5245696697285b40e216450caa63979d1ed838, we checked the correctness of the latest fixes and more documentation was added.

At the fourth iteration, the code was moved to the doubledice-token git repository, with a single commit (`0cd8145daa227b559cc44043347e70ddb07eabb4`). See the Names changed in the fourth iteration section for details on the changes made for this iteration.

At last, in the fifth iteration the life cycle of the Vesting contract was improved. The corresponding commit is `5edc0aeac04a427d4071213554aa20c121e898d4`.

## Contracts

The audited contracts are:

- `/contracts/DoubleDiceToken.sol`: ERC20 compatible token that generates yields. This contract depends on the following contracts that were also analyzed as part of the audit:

    - `/contracts/base/AbstractYields.sol`: Abstract contract that contains the logic used to award yields (removed before the second iteration).
    - `/contracts/base/ERC20Yields.sol`: abstract contract used to make an ERC20 token that uses the `AbstractYields.sol` logic (removed before the second iteration).
- `/contracts/DoubleDiceTokenVesting.sol`: contract used to vest tokens. See appendix A for our understanding of the vesting procedure. This file was named `TokenVesting.sol` in the first three iterations.
- `/contracts/DoubleDiceTokenVestingProxyFactory.sol`: factory for the DoubleDiceTokenVesting contracts. Added in the fifth iteration.

## Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

## Findings and Fixes

| ID | Title | Severity | Status |
|---|---|---|---|
| CR-01 | Unrestricted Burning of Tokens by Contract Owner | Critical | Fixed |
| CR-02 | Race Condition in DoubleDiceTokenVesting Allows DoubleDice to Claim The Initially Claimable Amount Instead of The User | Critical | Fixed |
| CR-03 | Yields Distribution Locks Tokens in DoubleDiceToken | Critical | Fixed |
| ME-01 | Unwanted transaction reversion in DoubleDiceTokenVesting.removeTokenGrant | Medium | Fixed |
| ME-02 | Initially Claimable Amount Can Be Collected Before startTime in DoubleDiceTokenVesting | Medium | Fixed |
| ME-03 | Integer Overflows in DoubleDiceToken | Medium | Fixed |
| ME-04 | Error Calculating Yield To Collect in DoubleDiceTokenVesting.collectYield | Medium | Fixed |
| ME-05 | Possible denial of service if too many tokens are excluded when DoubleDiceToken.distributeYield() is called | Critical | Fixed |
| MI-01 | Pragmas Not Locked to Specific Compiler Version | Minor | Fixed |
| MI-02 | Unnecessary Reentrancy in DoubleDiceTokenVesting when Calling token.safeTransfer() | Minor | Fixed |
| MI-03 | Denial of Service in DoubleDiceTokenVesting.removeTokenGrant() | Minor | Mitigated |
| MI-04 | Reversion With Wrong Error Message in DoubleDiceTokenVesting.addTokenGrant() | Minor | Fixed |
| MI-05 | Potential unwanted transaction revertions in DoubleDiceTokenVesting | Minor | Fixed |
| MI-06 | Integer overflow while checking for integer overflow in DoubleDiceToken | Minor | Fixed |

| | | | |
|---|---|---|---|
| EN-01 | SafeMath Libraries Removal | Enhancement | Fixed |
| EN-02 | transferFrom Used When is not Required in DoubleDiceToken | Enhancement | Fixed |
| EN-03 | Better API to Get Tokens Out of Vesting | Enhancement | Fixed |
| EN-04 | Give Yield on DoubleDiceTokenVesting.removeTokenGrant() | Enhancement | Fixed |
| EN-05 | Better DoubleDiceTokenVesting Life Cycle | Enhancement | Fixed |

## Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

| SEVERITY | EXPLOITABLE | ROADBLOCK | TO BE FIXED |
|---|---|---|---|
| Critical | Yes | Yes | Immediately |
| Medium | In the near future | Yes | As soon as possible |
| Minor | Unlikely | No | Eventually |
| Enhancement | No | No | Eventually |

# Issues Found by Severity

## Critical severity

### CR-01 Unrestricted Burning of Tokens by Contract Owner

The implementation of the DoubleDiceToken allows the token contract owner to burn someone else's tokens without their consent.

This is the function `DoubleDiceToken.burn()` (lines 18-20 of `contracts/DoubleDiceToken.sol`):

```
function burn(address from, uint256 amount) external onlyOwner {
  _burn(from, amount);
}
```

This function can be called by the owner of the contract to burn any amount of tokens from any user without restrictions. This is not expected by the users. The development team has confirmed that this is not the expected behavior and hence it must be corrected. Burn should respect allowances as defined in ERC20.

#### Recommendation

In order to fix this problem you can use the `ERC20Burnable` contract. It can be found in `@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol`

#### Solution

Fixed for the second iteration. The contract owner can only burn non-distributed tokens.

### CR-02 Race Condition in DoubleDiceTokenVesting Allows DoubleDice to Claim The Initially Claimable Amount Instead of The User

There are situations where DoubleDice may end up getting the initial claimable tokens of a vesting instead of these tokens going to the vesting user.

See the `DoubleDiceTokenVesting.removeTokenGrant()` function (lines 98-118 of `contracts/DoubleDiceTokenVesting.sol`)

```
function removeTokenGrant() external
onlyOwner
{
    uint32 monthsVested;
    uint256 amountVested;
```

```
        (monthsVested, amountVested) = calculateGrantClaim();
        uint256 amountNotVested =
(tokenGrant.amount.sub(tokenGrant.totalClaimed).sub(amountVested)).toUint128();

        token.safeTransfer(userAddress, amountVested);
        token.safeTransfer(dodiOwner, amountNotVested);

        tokenGrant.startTime = 0;
        tokenGrant.amount = 0;
        tokenGrant.vestingDuration = 0;
        tokenGrant.vestingCliff = 0;
        tokenGrant.monthsClaimed = 0;
        tokenGrant.totalClaimed = 0;
        tokenGrant.initiallyClaimableAmount = 0;

        emit GrantRemoved(amountVested, amountNotVested);
    }
```

This function calculates the amount vested and sends it to the user, while the grant amount left is sent to `dodiOwner`, which is set in the contract constructor. However, if the user has not claimed the initially claimable amount before the grant is removed, it is sent to `dodiOwner` instead of to the user. The user's resulting balance will depend on whether they claim that amount before the vesting owner calls `DoubleDiceTokenVesting.removeTokenGrant()` or not.

### Recommendation

Function `DoubleDiceTokenVesting.removeTokenGrant()` should be modified to take into account the initially claimable amount.

### Solution

Fixed for the second iteration. Now the initial claimable amount is transferred to the user if it was not transferred before when executing the `DoubleDiceTokenVesting.removeTokenGrant()` function.

## CR-03 Yields Distribution Locks Tokens in DoubleDiceToken

Each time the `DoubleDiceToken.distribute()` function (in `/contracts/DoubleDiceToken.sol`, lines 31-34) is called some of the tokens get locked in the contract address and can never be moved out.

```
function distribute(uint256 amount) external {
    transferFrom(msg.sender, address(this), amount);
   _distributeYield(amount);
}
```

Each time this function is called, the contract itself is eligible to get yields. Therefore, those withdrawable tokens are locked in the `DoubleDiceToken` contract and there is no mechanism to distribute them. Over time, the amount of token circulating will be reduced exponentially.

See appendix B for actual exploitation code.

### Recommendation
The model underlying the distribution and collection of yields should be documented and checked for consistency.

### Solution
Fixed by the DoubleDice token redesign in the second iteration. The mechanism to distribute yields was documented in `Token.md` for the third iteration.

## Medium severity
### ME-01 Unwanted transaction reversion in DoubleDiceTokenVesting.removeTokenGrant()
In line 104 of `contracts/DoubleDiceTokenVesting.sol`

```
uint256 amountNotVested =
(tokenGrant.amount.sub(tokenGrant.totalClaimed).sub(amountVested)).toUint128();
```

The value is cast to a 128 bits unsigned integer even when the result is stored in a 256 bits unsigned integer and the variables involved in the calculation are all 256 bits unsigned integers. This may be a problem if `amountNotVested` exceeds $2^{128}$, resulting in a reversion of the transaction.

### Recommendation
Remove the toUint128 call.

### Solution
Fixed for the second iteration.

### ME-02 Initially Claimable Amount Can Be Collected Before `startTime` in DoubleDiceTokenVesting
If `startTime` is set in the future, the `DoubleDiceTokenVesting.collectInitiallyClaimableAmount()` function (lines 135-145 of `contracts/DoubleDiceTokenVesting.sol`) can be fully executed, before the set `startTime`, transfering the initially claimable amount to the user.

### Recommendation

Add a `require(currentTime() >= tokenGrant.startTime);` instruction at the beginning of the function to solve this issue.

### Solution

Fixed for the second iteration.

## ME-03 Integer Overflows in DoubleDiceToken

If too many tokens are minted, the `AbstractYields` contract (from which the `DoubleDiceToken` contract inherits) may overflow when doing `distribute()` and `collect()`, which may make it impossible to either distribute yields to all users or collect tokens for a single user. This happens because the transaction reverts (caused by an integer overflow).

Integer overflow reverts are triggered in two places:

- at `AbstractYields._distributeYield (contracts/base/AbstractYields.sol:98)`
- at `AbstractYields.cumulativeYieldOf (contracts/base/AbstractYields.sol:75)`

This issue can be triggered only if the total supply of DoubleDice tokens exceeds $2^{127}$. See appendix C for actual exploitation code.

### Recommendation

Given that the DoubleDice token is minted only when constructing, adding a require statement in the `DoubleDiceToken` constructor (lines 9-16 of `contracts/DoubleDiceToken.sol`) to make the minted tokens to be less than $2^{127}$ should be enough to solve this issue.

### Solution

Fixed by the DoubleDice token redesign in the second iteration. Other changes regarding integer overflows were made for the third iteration.

## ME-04 Error Calculating Yield To Collect in DoubleDiceTokenVesting.collectYield

After claiming vested tokens, if yields are collected, the amount of yields collected is less than the yields generated.

The DoubleDiceTokenVesting.`collectYield()` function, which can be found in lines 180-185 of `/contracts/DoubleDiceTokenVesting.sol`, should send to the

user the accrued yield from the vested tokens. The yield to send should be calculated by deducting them from the contract balance, the grant amount and the claimed amount. The actual implementation considers only the grant amount and the current balance. Therefore, after claiming the vested tokens the first time, `DoubleDiceTokenVesting.collectYield()` will transfer zero or less than the expected.

#### Recommendation

Include `tokenGrant.totalClaimed` into the yield calculation for a more accurate result.

#### Solution

Fixed for the second iteration.

### ME-05 Possible denial of service if too many tokens are excluded when DoubleDiceToken.distributeYield() is called

Issue found in the second iteration.

If the addresses passed to the `excludedAccounts` parameter of the `DoubleDiceToken.distributeYield()` have a balance that exceeds (counting it each time the function is called) the `totalYieldAmount` passed in the contract constructor, the contract may raise an integer overflow in lines 148 (in the same function) or 63 (in function `balancePlusUnclaimedYieldOf()`) of `DoubleDiceToken.sol` rendering the yield distribution mechanism broken.

#### Recommendation

Limit the addresses excluded in the `DoubleDiceToken.distributeYield()` function so this cannot happen.

#### Solution

This was solved for the third iteration.

## Minor severity

### MI-01 Pragmas Not Locked to Specific Compiler Version

All `.sol` files in repo use the `pragma "solidity ^0.8.0"`.

#### Recommendation

It is better to lock to a specific compiler version (for instance `"solidity 0.8.6"`, to be consistent with `hardhat.config.ts`). See

https://consensys.github.io/smart-contract-best-practices/recommendations/#lock-pragmas-to-specific-compiler-version for details.

### Solution
Fixed for the second iteration.

## MI-02 Unnecessary Reentrancy in DoubleDiceTokenVesting when Calling token.safeTransfer()

The functions `DoubleDiceTokenVesting.addTokenGrant()` (line 82), `DoubleDiceTokenVesting.removeTokenGrant()` (lines 106 and 107) and `DoubleDiceTokenVesting.collectInitiallyClaimableAmount()` (line 139) allow reentrancy when calling `token.safeTransfer()`, which is arbitrary code in the blockchain. See https://docs.soliditylang.org/en/v0.8.6/security-considerations.html#use-the-checks-effects-interactions-pattern for details. Another option to solve this problem is to use a reentrancy guard.

The severity of this issue was reduced because we expect the token to be a DoubleDice token.

### Recommendation
In order to solve this problem, move the calls to the end of the functions, just above the emit calls.
The vesting contract makes calls to an unknown token, which is arbitrary code in the blockchain. This is not necessarily a problem, as token holders are presumed to trust the token contract itself and any damage that it may do will only affect itself. But this will make the reentrancy equivalent to a subsequent call, and removes all possibilities of exploiting intermediary states and it is standard practice.

Another option to solve this issue is to add a reentrancy guard.

### Solution
Fixed for the second iteration.

## MI-03 Denial of Service in DoubleDiceTokenVesting.removeTokenGrant()

When a grant is removed, the vested tokens are sent to the user and the rest is sent back to its original owner. Two transfer calls are involved in this function (lines 106 and 107 of `contracts/DoubleDiceTokenVesting.sol`). If one of them fails, the other is not made.

The severity of this issue was reduced because we expect the token to be a DoubleDice token.

### Recommendation

Use the "Favor pull over push for external calls" pattern to solve this problem. See https://consensys.github.io/smart-contract-best-practices/recommendations/#favor-pull-over-push-for-external-calls for details.

### Solution

The development team informed us that they prefer to have this potential issue, because it will never trigger when the `DoubleDiceTokenVesting` contract is deployed using the `DoubleDice` token, and it will consume less gas as it is now.

## MI-04 Reversion With Wrong Error Message in DoubleDiceTokenVesting.addTokenGrant

If `_amount < _initiallyClaimableAmount` the function `DoubleDiceTokenVesting.addTokenGrant()` will fail in line 77 of `contracts/DoubleDiceTokenVesting.sol`

```
uint amountVestedPerMonth = (_amount - _initiallyClaimableAmount) / _vestingDuration;
```

instead of line 79

```
require(_initiallyClaimableAmount < _amount, "initial claimable should be less than the total amount");
```

### Recommendation

Move line 79 above line 77 to solve the issue; or remove line 79 for a very slight gas usage reduction but get a worse error message.

### Solution

Fixed for the second iteration.

## MI-05 Potential unwanted transaction revertions in DoubleDiceTokenVesting

Issue found in the second iteration.

Even though `tokenGrant.totalClaimed` is an uint256 in the DoubleDiceTokenVesting contract, its calculation is casted to `uint128` in functions `claimVestedTokens()` (line 147 of `DoubleDiceTokenVesting.sol`) and `collectInitiallyClaimableAmount()` (line 159 of

`DoubleDiceTokenVesting.sol`). This may raise an exception if the calculated total claim exceeds $2^{128}$.

The severity of this issue was lowered because it is not possible to make this overflow with a DoubleDice token given the restrictions enforced in the `DoubleDiceToken._checkForPotentialOverflow()` function (lines 43-47 of `DoubleDiceToken.sol`). It might be possible if the ONE constant is changed in the future or if the `DoubleDiceTokenVesting` contract is used with a different token.

#### Recommendation

Do not cast the calculation results to `uint128`.

#### Solution

This issue was solved for the third iteration.

### MI-06 Integer overflow while checking for integer overflow in DoubleDiceToken

Issue found in the second iteration.

The `_checkForPotentialOverfow()` function of the `DoubleDiceToken` contract will raise an integer overflow exception if `initTotalSupply` >= $2^{127}$. The offending expression is `4 * initTotalSupply**2`.

#### Solution

This issue was solved for the third iteration when the overflow prevention was re-written.

## Enhancements

### EN-01 SafeMath Libraries Removal

Given that the solidity version is >= 0.8 there is no need to use SafeMath libraries. Usage of `"../libraries/LowGasSafeMath.sol"` and `"@openzeppelin/contracts/utils/math/SignedSafeMath.sol"` is not required. See https://docs.soliditylang.org/en/v0.8.6/080-breaking-changes.html#how-to-update-your-code for details (section "How to update your code").

#### Solution

Unneeded safe-math-libraries usage was removed for the second iteration. Only a single use of `SafeMath.tryMul()` was left, and that usage was not covered by

solidity 0.8, so it is ok. Given the code changes made for the third iteration, this usage was also removed.

### EN-02 transferFrom Used When is not Required in DoubleDiceToken

For the `DoubleDiceToken.distribute()` function, the `DoubleDiceToken` contract calls `transferFrom()` with `msg.sender` always as the first argument. Unnecessarily, it requires the users to give allowance to themselves.

#### Recommendation

Call to `transfer()` instead. It removes the allowance requirement.

#### Solution

Fixed for the second iteration.

### EN-03 Better API to Get Tokens Out of Vesting

Instead of having 3 different functions to extract tokens in the `DoubleDiceTokenVesting` contract (`claimVestedTokens()`, `collectInitiallyClaimableAmount()`, `collectYield()`), have a single parameterless `DoubleDiceTokenVesting.claim()` method callable by the user that transfers to the user all the tokens available at the moment of the call.

#### Solution

A `claim()` function with the suggested functionality was added for the second iteration, but the original functions are still in the source code. This is OK, the development team chose to give the option of a more complicated API, which allows the users to save some gas.

### EN-04 Give Yields on DoubleDiceTokenVesting.removeTokenGrant()

Change `DoubleDiceTokenVesting.removeTokenGrant()` code to give the accrued yields to the user when it is executed.

#### Solution

Fixed for the second iteration.

### EN-05 Better DoubleDiceTokenVesting Life Cycle

We recommend to:

- Fuse the constructor of the `DoubleDiceTokenVesting` contract and the `DoubleDiceTokenVesting.addTokenGrant()` method.

- Make the `DoubleDiceTokenVesting.removeTokenGrant()` method `selfdestruct` the contract
    - In order to do this, EN-04 is required

Solution
The life cycle of the vesting contract was improved by the development team for the fifth iteration. See the `EN-05.md` in git for details.

# Security considerations

## No-overflow warranties in DoubleDiceToken

The `DoubleDiceToken` contract (in `DoubleDiceToken.sol`) would overflow `uint256` if not treated carefully. In order to do that, limits to the amount of tokens minted, amount of tokens distributed as yields, tokens to be skipped from distribution were set, and the scale used to do the calculations (the `ONE` constant). Those limits were carefully considered because they allowed to reduce the yields lost to integer arithmetic. The details of the mathematical demonstration of those limits can be found in `Tokens.md` and also a python script `calculate_DoubleDiceToken_ONE.py` to assist in the calculation of the actual values of those constants. Those files were added for the third iteration.

## Yield-distribution exclusions in DoubleDiceToken

In the second and third iterations of this audit, the `DoubleDiceToken.distributeYield()` function (in lines 125-161 of `DoubleDiceToken.sol`) allows the token-contract owner to exclude some addresses from the yield distribution. In `Tokens.md` it is explained that this functionality will be used to not distribute yields to DEX contracts that are not prepared to receive them. But it has to be noted that this is a possible centralization excess. This functionality potentially allows the token-contract owner to exclude any address of the yield distribution process.

## Names changed in the second iteration

For the second iteration of this audit several functions and variables had their name changed to use `yield` to refer to the yields generated by the DODI token. As requested by the client we will refer to them using the new names in the audit, even when referring to the first iteration.

We consider that these changes do not change the results of this audit.

### Names changed in the fourth iteration

For the fourth iteration of this audit, several names have been changed, including filenames, contract names and variable names to refer to the current token name. As requested by the client we will refer to them using the new names in the audit.

We consider that these changes do not change the results of this audit.

## Conclusion

In the first iteration of this audit, three critical issues were found, in both DoubleDiceToken and DoubleDiceTokenVesting contracts. Additionally four medium-severity issues were found in both contracts. Several other minor issues and possible enhancements were also raised.

Most issues were solved for the second iteration, including a redesign of the DoubleDiceToken contract. The only issue not solved was MI-03 for which the development team chose not to solve it because they claim that settings outside the scope of the audit ensure it will never trigger.

Some new issues were found and a third iteration was required. In the second iteration we found 1 medium-severity issue (ME-05) and another 2 minor ones. All those issues were fixed for the third iteration, and no additional issues were found.

Also some security considerations were documented in the "Security considerations" section.

**Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the DoubleDice project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.**

# Appendixes

## Appendix A: Vesting procedure

The vesting schedule coded in `/contracts/DoubleDiceTokenVesting.sol`, as we understand it, has the following properties:

- An initial claim (configurable) can be obtained by the user after the starting date.
- New vested tokens are awarded every month, starting at the start date and for the entire duration of the vesting process.
    - Each month, the same amount of tokens is awarded
- Vested tokens can be claimed after the vesting cliff, including those not claimed in previous months.
- The owner of the vesting contract can stop the vesting process.
- If the vesting process is stopped by the vesting owner, all the tokens that can be claimed by the user are transferred to them and the rest is transferred back to the DoubleDice owner.
- Yields generated by the tokens involved in the vesting scheme can be collected by the user.
    - If the vesting contract is stopped, the yields generated by the vesting contract must be collected by the user and they do not go back to the DoubleDice owner.

# Appendix B: Code to trigger supply reduction in DoubleDiceToken

Hereunder is the code we used to trigger CR-03 (Yields Distribution Locks Tokens in DoubleDiceToken).

```
import { ethers, waffle } from "hardhat";
import { expect } from "chai";
import { DoubleDiceToken } from "../typechain";
import { toBigNumber } from "./utils";

describe("AuditTest", () => {
 let [wallet, wallet1, wallet2, tokenHolder] = waffle.provider.getWallets();
 let erc20: DoubleDiceToken;

 beforeEach("Deploy DoubleDiceToken", async () => {
   const factory = await ethers.getContractFactory("DoubleDiceToken");
   const totalSupply = toBigNumber(100);
   erc20 = (await factory.deploy(
       totalSupply,
     "DoubleDice Token",
     "DODI",
     tokenHolder.address
   )) as DoubleDiceToken;
 });

 describe("distribute", () => {
   it("with just one wallet", async () => {
     await (
       await erc20.connect(tokenHolder).approve(tokenHolder.address, toBigNumber(20))
     ).wait();
     await (await erc20.connect(tokenHolder).distribute(toBigNumber(20))).wait();

     expect(await erc20.balanceOf(erc20.address)).to.eq(toBigNumber(20));

     expect(await erc20.cumulativeYieldOf(tokenHolder.address)).to.eq(toBigNumber(20));

     await (await erc20.connect(tokenHolder).collect()).wait();
     expect(await erc20.balanceOf(tokenHolder.address)).to.eq(toBigNumber(100));
   });
 });
});
```

Add a `DoubleDiceTokenLockTokens.spec.ts` file with this content to the `test` folder to run these tests (which will fail showing the overflows).

This is the failing stack trace:

```
  1) AuditTest
       distribute
         with just one wallet:

      AssertionError: Expected "16000000000000000000" to be equal
20000000000000000000
      + expected - actual

       {
      -  "_hex": "0x01158e460913d00000"
      +  "_hex": "0xde0b6b3a76400000"
         "_isBigNumber": true
       }
      at
/home/aure/devel/doubledice-token/test/DoubleDiceTokenLockTokens.spe
c.ts:30:73
      at step (test/DoubleDiceTokenLockTokens.spec.ts:33:23)
      at Object.next (test/DoubleDiceTokenLockTokens.spec.ts:14:53)
      at fulfilled (test/DoubleDiceTokenLockTokens.spec.ts:5:58)
      at processTicksAndRejections
(internal/process/task_queues.js:95:5)
      at runNextTicks (internal/process/task_queues.js:64:3)
      at listOnTimeout (internal/timers.js:526:9)
      at processTimers (internal/timers.js:500:7)
```

If you comment out the failing `expect`, the test will also fail on the final line.

This test demonstrates the supply reduction for just one token holder besides the token contract. This can be replicated for more complex cases with more players.

# Appendix C: Code to trigger integer overflows in DoubleDiceToken

Hereunder is the code we used to trigger ME-03 ("Integer overflows in DoubleDiceToken")

```
import {ethers, waffle} from 'hardhat';
import {DoubleDiceToken} from '../typechain';
import {BigNumber} from 'ethers';

describe('DoubleDiceToken overflow issues', () => {
   const [owner] = waffle.provider.getWallets();
   let DoubleDiceToken: DoubleDiceToken;
   const name = 'DoubleDice Token';
   const symbol = DODI;
   const totalSupply = BigNumber.from("1393796574908163946345982392040522594123776"); //
2**140

   beforeEach('Deploy DoubleDiceToken', async () => {
       const factory = await ethers.getContractFactory('DoubleDiceToken');
       DoubleDiceToken = (await factory.deploy(totalSupply, name, symbol, owner.address))
as DoubleDiceToken;
   });

   it('has problems when distributing more than 2**128', async () => {
       const distributed = "340282366920938463463374607431768211458"; // 2**128 + 2
       await DoubleDiceToken.approve(owner.address, distributed);
       await DoubleDiceToken.distribute(distributed);
   });

   it('has problems when collecting more than 2**127', async () => {
       const distributed = "170200000000000000000000000000000000000"; // slightly over
2**127
       await DoubleDiceToken.approve(owner.address, distributed);
       await DoubleDiceToken.distribute(distributed);
       await DoubleDiceToken.collectFor(owner.address);
   });
});
```

Add a `DoubleDiceTokenOverflows.spec.ts` file with this content to the `test` folder to run these tests (which will fail showing the overflows).

These are the failing stack traces:

```
  1) DoubleDiceToken overflow issues
       has problems when distributing more than 2**128:
     Error: Transaction reverted and Hardhat couldn't infer the
reason. Please report this to help us improve Hardhat.
```

```
      at DoubleDiceToken.mul
(contracts/libraries/LowGasSafeMath.sol:54)
      at DoubleDiceToken.internal@5351
(contracts/DoubleDiceToken.sol)
      at DoubleDiceToken.mul
(contracts/libraries/LowGasSafeMath.sol:54)
      at DoubleDiceToken._distributeYield
(contracts/base/AbstractYields.sol:98)
      at DoubleDiceToken.distribute
(contracts/DoubleDiceToken.sol:33)
      at DoubleDiceToken.distribute
(contracts/DoubleDiceToken.sol:31)
      at runMicrotasks (<anonymous>)
      at processTicksAndRejections
(internal/process/task_queues.js:95:5)
      at runNextTicks (internal/process/task_queues.js:64:3)
      at listOnTimeout (internal/timers.js:526:9)
      at processTimers (internal/timers.js:500:7)
      at HardhatNode._mineBlockWithPendingTxs
(node_modules/hardhat/src/internal/hardhat-network/provider/node.ts:
1154:23)
      at HardhatNode.mineBlock
(node_modules/hardhat/src/internal/hardhat-network/provider/node.ts:
377:16)

  2) DoubleDiceToken overflow issues
       has problems when collecting more than 2**127:
     Error: VM Exception while processing transaction: revert
SafeCast: value doesn't fit in an int256
      at DoubleDiceToken.transferFrom
(@openzeppelin/contracts/token/ERC20/ERC20.sol:152)
      at DoubleDiceToken.cumulativeYieldOf
(contracts/base/AbstractYields.sol:73)
      at DoubleDiceToken.cumulativeYieldOf
(contracts/base/AbstractYields.sol:74)
      at DoubleDiceToken.withdrawableYieldsOf
(contracts/base/AbstractYields.sol:53)
      at DoubleDiceToken._prepareCollect
(contracts/base/AbstractYields.sol:110)
      at DoubleDiceToken.collectFor
```

```
(contracts/DoubleDiceToken.sol:23)
      at DoubleDiceToken.collectFor
(contracts/DoubleDiceToken.sol:22)
      at runMicrotasks (<anonymous>)
      at processTicksAndRejections
(internal/process/task_queues.js:95:5)
      at runNextTicks (internal/process/task_queues.js:64:3)
      at listOnTimeout (internal/timers.js:526:9)
      at processTimers (internal/timers.js:500:7)
      at HardhatNode._mineBlockWithPendingTxs
(node_modules/hardhat/src/internal/hardhat-network/provider/node.ts:
1154:23)
      at HardhatNode.mineBlock
(node_modules/hardhat/src/internal/hardhat-network/provider/node.ts:
377:16)
```