Contribute DeFi

Security Assessment

October 13th, 2020

By :
Camden Smallwood @ CertiK
camden.smallwood@certik.org

Disclaimer

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has indeed completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	Contribute DeFi
Description	A decentralized capital coordination tool that incentivizes the creation of a perpetual interest-generating pool through smart contracts on the Ethereum blockchain.
Platform	Ethereum; Solidity
Codebase	GitHub Repository
Commits	1. 7478d34e82eaa5cb9d2f692c4731ce687a000392

Audit Summary

Delivery Date	Oct. 13, 2020
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	2
Timeline	Oct. 5, 2020 - Oct. 9 2020

Vulnerability Summary

Total Issues	18
Total Critical	0
Total Major	0
Total Minor	1
Total Informational	17

Executive Summary

Preliminary:

The codebase was found to be well-defined, has proper access restriction where necessary and includes expansive documentation and commenting throughout. While the math in the codebase was found to be sound in regards to the specification, we approached the code with due dilligence and identified multiple optimizations that could be applied. The optimizations are not the result of any compromises to the system, but would help to improve the overall performance and readability of the code. Primarily the `MathUtils.sqrt` function is unsafe due to allowing overflow, but most common were explicit usage of `SafeMath` functions, explicitly returning local variables, and inefficient `uint256` comparisons in relation to zero. Higher gas costs on Ethereum are an increasing issue and should always be taken into consideration when designing a new Smart Contract system.

Alleviations:

All of the optimizations were taken into consideration by the team, whose strong focus on security took precedence over fined-tuned code optimization. The team communicated that the keys were possibly burnt, which prevents the optimizations from being applied to the deployed versions of the contracts, but that they will take everything into consideration as new contract different layers are built.

Findings

ID	Title	Type	Severity	Status
TRIB-01	Integer overflow in `MathUtils.sqrt` function	Arithmetic	Minor	Acknowledged
TRIB-02	Inefficient uint256 comparison with zero constant	Performance	Informational	Acknowledged
TRIB-03	Non-optimal `Contribute.claimInterest` implementation	Performance	Informational	Acknowledged
TRIB-04	`Contribute.claimRequired` function should be external	Implementation	Informational	Acknowledged
TRIB-05	Non-optimal `Contribute.getInterest` implementation	Performance	Informational	Acknowledged
TRIB-06	Unnecessary explicit usage of `SafeMath.div`	Implementation	Informational	Acknowledged
TRIB-07	Unnecessary explicit usage of `SafeMath` functions	Implementation	Informational	Acknowledged
TRIB-08	Non-optimal `Contribute._invest` implementation	Performance	Informational	Acknowledged
TRIB-09	Non-optimal `Contribute._sell` implementation	Performance	Informational	Acknowledged
TRIB-10	Non-optimal `Contribute._calculateClaimableAmount` implementation	Performance	Informational	Acknowledged
TRIB-11	Explicitly returning local variable	Implementation	Informational	Acknowledged
TRIB-12	Explicitly returning local variable	Implementation	Informational	Acknowledged
TRIB-13	Explicitly returning local variable	Implementation	Informational	Acknowledged
TRIB-14	Inefficient uint256 comparison with zero constant	Performance	Informational	Acknowledged
TRIB-15	`Genesis.totalReserveBalance` function should be external	Implementation	Informational	Acknowledged
TRIB-16	Non-optimal `Vault.deposit` implementation	Performance	Informational	Acknowledged
TRIB-17	Non-optimal `Vault.redeem` implementation	Performance	Informational	Acknowledged
TRIB-18	Non-optimal `Vault.getBalance` implementation	Performance	Informational	Acknowledged

TRIB-01: Integer overflow in `MathUtils.sqrt` function

Type	Severity	Location
Arithmetic	Minor	utils/MathUtils.sol L10-L17

Description:

The `sqrt` function in the `MathUtils` library implements the Babylonian method for calculating the square root of a supplied `uint256 x` parameter. The implementation uses an initial iteration value of `z = (x + 1) / 2` which can result in an integer overflow if `x` is `uint256(-1)` and allows for division by zero in the calculation of `z = (x / z + z) / 2`, which will cause the transaction to revert.

Recommendation:

While the value returned from the current `sqrt` implementation is valid for other values, it should be refactored in order to prevent against division by zero:

```
function sqrt(uint256 x) internal pure returns (uint256 y) {
    if (x > 3) {
        uint256 z = x / 2 + 1;
        y = x;
        while (z < y) {
            y = z;
            z = (x / z + z) / 2;
        }
    } else if (x != 0) {
        y = 1;
    }
}
```

TRIB-02: Inefficient uint256 comparison with zero constant

Type	Severity	Location
Performance	Informational	utils/MathUtils.sol L21

Description:

The `roundedDiv` function in the `MathUtils` library performs a greater-than comparison between a `uint256` variable and a zero constant. This is inefficient because `uint256` values cannot be negative and comparison operators have a higher gas cost than equality operators:

```
require(b > 0, 'div by 0');
```

Recommendation:

Consider converting the greater-than comparison into an inequality check in order to reduce the overall cost of gas:

```
require(b != 0, 'div by 0');
```

TRIB-03: Non-optimal `Contribute.claimInterest` implementation

Type	Severity	Location
Performance	Informational	Contribute.sol L121-L123

Description:

The `claimInterest` function in the `Contribute` contract retrieves the balance of the message sender twice and calculates the total claim required twice, which is inefficient:

```
uint256 totalToClaim = token.balanceOf(msg.sender) < totalClaimRequired()
  ? token.balanceOf(msg.sender)
  : totalClaimRequired();
```

Recommendation:

Consider storing the balance of the message sender and the total claim require in local variables in order to reduce the overall cost of gas:

```
uint256 balance = token.balanceOf(msg.sender);
uint256 total = totalClaimRequired();
uint256 amount = balance < total ? balance : total;
```

TRIB-04: `Contribute.claimRequired` function should be external

Type	Severity	Location
Implementation	Informational	Contribute.sol L136

Description:

The `claimRequired` function in the `Contribute` contract is declared `public`, yet it is not used within the `Contribute` contract itself. This is inefficient because `public` functions have a higher gas cost `external` functions:

```
function claimRequired(uint256 amountToClaim) public view returns (uint256)
```

Recommendation:

Consider converting the function from `public` to `external` in order to reduce the overall cost of gas:

```
function claimRequired(uint256 amountToClaim) external view returns (uint256)
```

TRIB-05: Non-optimal `Contribute.getInterest` implementation

Type	Severity	Location
Performance	Informational	Contribute.sol L148-L156

Description:

The `getInterest` function in the `Contribute` contract declares and explicitly returns a local `uint256 vaultBalance` variable after performing a possibly unnecessary subtraction. This is inefficient because returned variables can be declared in the function signature (which allows for omitting the return statement) and if the vault balance is known to be the same as the total reserve, then subtraction is unnecessary.

Recommendation:

Consider declaring a `uint256 interest` variable in the function signature, omitting the explicit return statement and performing a primitive subtraction instead of using the `SafeMath.sub` function in order to reduce the overall cost of gas, but only if the vault balance is greater than the total reserve:

```
function getInterest() public view returns (uint256 interest) {
  uint256 vaultBalance = IVault(vault).getBalance();
  if (vaultBalance > totalReserve) {
    interest = vaultBalance - totalReserve;
  }
}
```

TRIB-06: Unnecessary explicit usage of `SafeMath.div`

Type	Severity	Location
Implementation	Informational	Contribute.sol L186

Description:

The `getReserveToTokensTaxed` function in the `Contribute` contract contains unnecessary explicit usage of the `SafeMath.div` function:

```
uint256 fee = SafeMath.div(reserveAmount, TAX);
```

Recommendation:

Since `SafeMath` is being used in the `Contribute` contract for `uint256` types, consider refactoring to use the `SafeMath.div` function as a function call to the `reserveAmount` variable:

```
uint256 fee = reserveAmount.div(TAX);
```

TRIB-07: Unnecessary explicit usage of `SafeMath` functions

Type	Severity	Location
Implementation	Informational	Contribute.sol L200-L201

Description:

The `getTokensToReserveTaxed` function in the `Contribute` contract contains unnecessary explicit usage of the `SafeMath.div` and `SafeMath.sub` functions:

```
uint256 fee = SafeMath.div(reserveAmount, TAX);
return SafeMath.sub(reserveAmount, fee);
```

Recommendation:

Since `SafeMath` is being used in the `Contribute` contract for `uint256` types, consider refactoring to use the `SafeMath.div` and `SafeMath.sub` functions as function calls on the `reserveAmount` variable:

```
uint256 fee = reserveAmount.div(TAX);
return reserveAmount.sub(fee);
```

TRIB-08: Non-optimal `Contribute._invest` implementation

Type	Severity	Location
Performance	Informational	Contribute.sol L224-L225, L231, L239, L241

Description:

The `_invest` function in the `Contribute` contract contains explicit calls to `SafeMath` functions, ignores the boolean value returned from the call to `IVault.deposit` and performs comparisons between `uint256` variables and constants in relation to zero, which is inefficient because `uint256` values cannot be negative and comparison operators have a higher gas cost than equality operators:

```
uint256 fee = SafeMath.div(_reserveAmount, TAX);
require(fee >= 1, 'Transaction amount not sufficient to pay fee');
```

```
require(taxedTokens > 0, 'This is not enough to buy a token');
```

```
IVault(vault).deposit(_reserveAmount);
```

```
totalReserve = SafeMath.add(totalReserve, _reserveAmount);
```

Recommendation:

Since `SafeMath` is being used in the `Contribute` contract for `uint256` types, consider refactoring to use the `SafeMath` functions as function calls on the variables directly, requiring the call to the `IVault.deposit` function to succeed, as well as converting the comparisons into inequality checks in order to reduce the overall cost of gas:

```
uint256 fee = _reserveAmount.div(TAX);
require(fee != 0, 'Transaction amount not sufficient to pay fee');
```

```
require(taxedTokens != 0, 'This is not enough to buy a token');
```

```
require(IVault(vault).deposit(_reserveAmount), 'Vault deposit failed');
```

```
totalReserve = totalReserve.add(_reserveAmount);
```

TRIB-09: Non-optimal `Contribute._sell` implementation

Type	Severity	Location
Performance	Informational	Contribute.sol L257, L260-L262, L264

Description:

The `_sell` function in the `Contribute` contract explicitly calls `SafeMath` functions and performs comparisons between `uint256` variables and constants in relation to zero, which is inefficient because `uint256` values cannot be negative and comparison operators have a higher gas cost than equality operators:

```
require(_tokenAmount > 0, 'Must sell something');
```

```
uint256 fee = SafeMath.div(reserveAmount, TAX);
require(fee >= 1, 'Must pay minimum fee');
```

```
uint256 net = SafeMath.sub(reserveAmount, fee);
```

```
totalReserve = SafeMath.sub(totalReserve, net);
totalInterestClaimed = SafeMath.add(totalInterestClaimed, claimable);
```

Recommendation:

Since `SafeMath` is being used in the `Contribute` contract for `uint256` types, consider refactoring to use the `SafeMath` functions as function calls on the variables directly, as well as converting the comparisons into inequality checks in order to reduce the overall cost of gas:

```
require(_tokenAmount != 0, 'Must sell something');
```

```
uint256 fee = reserveAmount.div(TAX);
require(fee != 0, 'Must pay minimum fee');
```

```
uint256 net = reserveAmount.sub(fee);
```

```
totalReserve = totalReserve.sub(net);
totalInterestClaimed = totalInterestClaimed.add(claimable);
```

TRIB-10: Non-optimal `Contribute._calculateClaimableAmount` implementation

Type	Severity	Location
Performance	Informational	Contribute.sol L305-L309

Description:

The `_calculateClaimableAmount` function in the `Contribute` contract performs an unnecessary equality check between a `uint256` variable and a zero constant before explicitly returning zero or that same `uint256` variable. This is inefficient because returned variables can be declared in the function signature (which allows for omitting the return statement), and the equality check is unnecessary because the function returns zero regardless.

Recommendation:

Consider declaring the `uint256 claimable` variable in the function signature and omitting the explicit return statement in order to reduce the overall cost of gas:

```
function _calculateClaimableAmount(uint256 _amount) internal view returns (uint256 claimable)
{
  uint256 interest = getInterest();
  claimable = _amount > interest ? interest : _amount;
}
```

TRIB-11: Explicitly returning local variable

Type	Severity	Location
Implementation	Informational	Contribute.sol L344-L345

Description:

The `_calculateReserveToTokens` function in the `Contribute` contract unnecessarily returns a local variable explicitly:

```
uint256 _supplyDelta = _newSupply.sub(_supply);
return _supplyDelta;
```

Recommendation:

Consider declaring the returned variable in the function signature on line 333:

```
) internal pure returns (uint256 _supplyDelta) {
```

Then adjust the local variable declaration and remove the explicit return statement on lines 344-345:

```
_supplyDelta = _newSupply.sub(_supply);
```

TRIB-12: Explicitly returning local variable

Type	Severity	Location
Implementation	Informational	Contribute.sol L364-L366

Description:

The `_calculateTokensToReserve` function in the `Contribute` contract unnecessarily returns a local variable explicitly:

```
uint256 _reserveDelta = _totalReserve.sub(_newReserve);
return _reserveDelta;
```

Recommendation:

Consider declaring the returned variable in the function signature on line 357:

```
) internal pure returns (uint256 _reserveDelta) {
```

Then adjust the local variable declaration and remove the explicit return statement on lines 364-366:

```
_reserveDelta = _totalReserve.sub(_newReserve);
```

TRIB-13: Explicitly returning local variable

Type	Severity	Location
Implementation	Informational	Contribute.sol L374-L379

Description:

The `_calculateReserveFromSupply` function in the `Contribute` contract unnecessarily returns a local variable explicitly:

```
return _reserve.roundedDiv(1e18); // correction of the squared unit
```

Recommendation:

Consider declaring the returned variable in the function signature on line 372, then adjust the local variable declaration and remove the explicit return statement on lines 364-366:

```
function _calculateReserveFromSupply(uint256 _supply) internal pure returns (uint256
_reserve) {
  // r = s^2 * m / 2
  _reserve = _supply
    .mul(_supply)
    .div(DIVIDER) // inverse the operation (Divider instead of multiplier)
    .div(2)
    .roundedDiv(1e18); // correction of the squared unit
}
```

TRIB-14: Inefficient uint256 comparison with zero constant

Type	Severity	Location
Performance	Informational	Genesis.sol L85

Description:

The `claim` function in the `Genesis` contract performs a greater-than comparison between a `uint256` value and a zero constant. This is inefficient because `uint256` values cannot be negative and comparison operators have a higher gas cost than equality operators:

```
require(balance[msg.sender] > 0, 'No tokens to claim');
```

Recommendation:

Consider converting the greater-than comparison into an inequality check in order to reduce the overall cost of gas:

```
require(balance[msg.sender] != 0, 'No tokens to claim');
```

TRIB-15: `Genesis.totalReserveBalance` function should be external

Type	Severity	Location
Implementation	Informational	Genesis.sol L110

Description:

The `totalReserveBalance` function in the `Genesis` contract is declared `public`, yet it only performs an external calls and is not used within the `Genesis` contract itself. This is inefficient because `public` functions have a higher gas cost `external` functions:

```
function totalReserveBalance() public view returns (uint256)
```

Recommendation:

Consider converting the function from `public` to `external` in order to reduce the overall cost of gas:

```
function totalReserveBalance() external view returns (uint256)
```

TRIB-16: Non-optimal `Vault.deposit` implementation

Type	Severity	Location
Performance	Informational	Vault.sol L44, L45

Description:

The `deposit` function in the `Vault` contract is declared `public`, yet it only performs external calls and is not used within the `Vault` contract itself. This is inefficient because `public` functions have a higher gas cost `external` functions:

```
function deposit(uint256 amount) public override returns (bool)
```

It also performs a greater-than comparison between a `uint256` parameter and a zero constant. This is inefficient because `uint256` values cannot be negative and comparison operators have a higher gas cost than equality operators:

```
require(amount > 0, 'Amount must be greater than 0');
```

Recommendation:

Consider converting the function from `public` to `external` and the greater-than comparison into an inequality check in order to reduce the overall cost of gas:

```
function deposit(uint256 amount) external override returns (bool)
```

```
require(amount != 0, 'Amount must be greater than 0');
```

TRIB-17: Non-optimal `Vault.redeem` implementation

Type	Severity	Location
Performance	Informational	Vault.sol L59, L60

Description:

The `redeem` function in the `Vault` contract is declared `public`, yet it only performs external calls and is not used within the `Vault` contract itself. This is inefficient because `public` functions have a higher gas cost `external` functions:

```
function redeem(uint256 amount) public override nonReentrant returns (bool)
```

It also performs a greater-than comparison between a `uint256` parameter and a zero constant. This is inefficient because `uint256` values cannot be negative and comparison operators have a higher gas cost than equality operators:

```
require(amount > 0, 'Amount must be greater than 0');
```

Recommendation:

Consider converting the function from `public` to `external` and the greater-than comparison into an inequality check in order to reduce the overall cost of gas:

```
function redeem(uint256 amount) external override nonReentrant returns (bool)
```

```
require(amount != 0, 'Amount must be greater than 0');
```

TRIB-18: Non-optimal `Vault.getBalance` implementation

Type	Severity	Location
Performance	Informational	Vault.sol L73-L81

Description:

The `getBalance` function in the `Vault` contract returns a `uint256` balance variable explicitly before performing an unnecessary greater-than comparison between that same `uint256` variable and a zero constant. This is inefficient because returned variables can be declared in the function signature (which allows for omitting the return statement), and the greater-than comparison is unnecessary due to the usage of the `SafeMath.mul` function on the following line of code.

Recommendation:

Consider declaring the `uint256 _balance` variable in the function signature, omitting the explicit return statement, and omitting the greater-than comparison (while will be handled by the `SafeMath.mul` function) in order to reduce the overall cost of gas:

```
function getBalance() public override view returns (uint256 _balance) {
  IMStable stableSavings = IMStable(savingsContract);
  _balance = stableSavings.creditBalances(address(this))
    .mul(stableSavings.exchangeRate())
    .div(1e18);
}
```