Solidified Crowdsale and Token Contracts Audit

v1.0

New Alchemy

June, 2018

Introduction

During June 2018, Solidified engaged New Alchemy to audit their Crowdsale and Token smart contracts for the `SOLID` Token.

The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts, finding differences between the contract's implementation and their behavior as described in public documentation, and finding any other issues with the contracts that may impact their trustworthiness. Solidified provided New Alchemy with access to a draft whitepaper and the GitHub repository containing the source code with an associated `README.md` file.

The audit was performed over 3 days. This document describes the issues discovered in the audit.

Disclaimer

The audit makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

Executive Summary

The Solidified contracts exhibited many positive characteristics, including extensive reuse of well known libraries, sufficiently and accurately commented source code (with one small exception) and the use of approximately 19 unit-level test cases. However, the audit identified a variety of findings with impacts ranging from moderate to very minor. There were no critical findings. The more significant findings involved:

- A complex logic bug in pre-purchase validation that may result in a stalled sale stage.
- Inconsistent and confusing declaration and behavior of critical constants.
- Integer math operation order leading to loss of precision.

The test cases were not strictly in scope but were utilized to help understand and confirm specific findings. Significant portions of the deployed code package were standard and copied from widely-used and reviewed OpenZeppelin[1] contracts. As a result, the primary focus of the review and findings pertain to the custom overrides and additions contained in the `TokenSale`, `SolidToken` and `Distributable` contract code. All payable functions are inherited through OpenZeppelin. The `README.md` documentation was considered primary.

[1] https://openzeppelin.org/ and https://github.com/OpenZeppelin/openzeppelin-solidity

Files Audited

The code reviewed by New Alchemy is in the GitHub repository https://github.com/solidified-platform/Crowdsale/tree/f46573fdbc2c79600e88a0a321ed40ca9ba84d76 which is at commit hash `f46573fdbc2c79600e88a0a321ed40ca9ba84d76`.

The specific files making up the complete deployed package include the following:

```
https://github.com/solidified-platform/Crowdsale/tree/f46573fdbc2c79600e88a0a321ed
        40ca9ba84d76/Smart-Contracts

  contracts/Distributable.sol             bd22baba9099604c3300efdb7682efaeeb58dca7
  contracts/SolidToken.sol                29a07d38dce27cd81d848cf8b7024f06574f0768
  contracts/TokenSale.sol                 f38aa21d236f486a3dc5c87dd21a4bdcf1ef0552


https://github.com/OpenZeppelin/openzeppelin-solidity/tree/e1dc1411fc154970216eb7c
        cc3855b1052f81889/contracts  (out of direct scope)

  crowdsale/Crowdsale.sol                 7118e62722e1e66391f350de751ff59316cf2c90
  crowdsale/emission/MintedCrowdsale.sol  2e65995d87af03627c6982ca82ab881f0a97485d
  crowdsale/validation/Whitelisted...sol  d7916f3f5f62f0bed8610d899cde5a42a5b1e75a
  lifecycle/Pausable.sol                  c8372f49d47f733ac4d057288181343f09a73e1a
  math/SafeMath.sol                       6c154b4e5d1c28b3757e6b850aab82371b2f75d1
  ownership/Ownable.sol                   4bc2f872b48b37adcfa9a9bca2aeab7cfa7d3591
  token/ERC20/BasicToken.sol              61b99e55729bd0715ee1e43495bf80d3e9e433c1
  token/ERC20/ERC20Basic.sol              46fdce538afcd0194a1d62ddcfa9cdd3c9ae55be
  token/ERC20/ERC20.sol                   dd9f2e2472db627af19d5cc66127c41e19e1bf61
  token/ERC20/MintableToken.sol           6582204becc7ed9223fc640ecc3a6247635e7a72
  token/ERC20/StandardToken.sol           6b6abd6be81e61e9cf9753f478f9d7403e3ff063
```

New Alchemy's audit was additionally guided by the whitepaper provided for this test as well as the README.md file and inspection of test files located in the repository.

General Discussion

The project consists of two primary parts. First, the `SolidToken` is intended to be a ERC20 mintable, ownable and standard token with a maximum supply of 4 million units, 18 decimal places per unit, a fixed exchange rate set at 1 SOLID = 0.015 ETH and a 6-month lock-in after the main sale concludes. Second, the `TokenSale` intends to implement a mintable, whitelisted, pausable and distributable contract with logic and constraints delineating two phases of a `SolidToken` sale.

The `SolidToken` is approximately 35 lines of code and imports a significant amount of functionality from OpenZeppelin[2]. The remainder of the code implements custom constants, functions related to token transfer date, and additional overridden functions for minting against a supply cap and enabling transfers.

The `TokenSale` (along with its supporting `Distributable`) is approximately 200 lines of code and also imports a large amount of functionality from OpenZeppelin. It contains many sale-related constants and staged-sale related functions. It overrides a significant amount of purchase validation functionality.

The `README.md` documentation describes what is expected to happen after the sale concludes, including additional minting. However, the implementation of this functionality was not observed in the code. The brief maximum participation `Whale protection` statement was difficult to connect to its exact implementation behavior. The sale code does implement functionality related to making change and enforcing a cap.

A number of positive aspects to the project were observed:

- Extensive reuse of well known libraries, e.g. OpenZeppelin
- Accurate commenting with the exception of `Distributable`
- Some unit-level test cases for the positive-scenario

Several general areas of concern developed around particular areas including:

- A complex logic bug in pre-purchase validation that may result in a stalled sale stage.
- Inconsistent and confusing declaration and behavior of critical constants.
- Minor discrepancies around critical constant units (e.g. days versus months)
- Minor loss of precision due to division before multiplication
- Minor language style: deprecated keywords, missing modifiers, and compiler warnings

There were some discrepancies between the `README.md` and white paper (e.g. Community & Audit Training Fund allocation, token burning). The presence and coverage of the unit testcases increases confidence of developer professionalism. New Alchemy believes the way forward should include further refinement of the documentation, corrections of the findings listed in this review and further test coverage development. The absence of critical findings is a very positive leading indicator.

[2] https://openzeppelin.org/ and https://github.com/OpenZeppelin/openzeppelin-solidity

Contract / Documentation Contract Coherence

This section describes how accurately the values from the `README.md` file are implemented in the actual contracts. Contracts should aim to implement as closely as possible the various descriptions found in the documentation and website. For this section, the `README.md` is going to be considered as the reference for all values. The amount of discrepancies found between the documentation and actual contracts help users decide the level of trust they can put into the contracts as they are implemented. The proper utilization and behavior of constants are addressed in subsequent sections.

Item name	Whitepaper Value	Smart contract Value	**Discrepancy (%)**
Total token supply (max)	4,000,000	4,000,000	**0%**
Token price	1 Solid = 0.015 ETH	set in constructor	**N/A**
Decimal places	18	18	**0%**
Pre-sale cap (max tokens)	1,600,000	set in constructor	**N/A**
Pre-sale duration	3 months max	90 days max	**~0%**
Pre-sale discount	20%	20%	**0**
Pre -> main break duration	10 days	10 days	**0%**
Main sale cap (total tokens)	2,400,0000	set in constructor	**N/A**
Main sale duration	1 month	30 days	**~0%**
Lock-in duration	6 months	182 days	**~0%**
Maximum participation	100 ETH	100 ETH	**0**
Minimum participation	0.5 ETH	0.5 ETH	**0**

Solidified did implement in the contracts many but not all of the critical constants detailed in the `README.md`. Some of the values trivially differed due to different units being specified, e.g. 3 months versus 90 days. However, several critical constants, including price and supply, are only set at contract creation time. This creates a strong burden on the token buyer to actually check these values on the blockchain to ensure the owner has set all the values as intended.

Importantly, the critical constants in `Distributable.sol` appear to be exclusively placeholders - the contracts never write to these values.

While New Alchemy acknowledges that it is easier for the contract owner to keep some of the values undefined until the contract is actually initiated/started, this lowers the overall confidence that a potential buyer can put into the contracts. New Alchemy strongly believes that high transparency standards strengthens the trust in the contracts and their owners. New Alchemy strongly recommends to clearly hardcode as many constants as possible and remove the inconsistencies between the contracts and documentation.

The `README.md` documentation makes several statements involving post-sale activity. These are not implemented in the contract. This may be due to a combination of scope, intent, inapplicability, separation of duties and/or the difficulty, complexity and cost of implementation.

Critical Issues

There were no critical issues found during the review.

Moderate Issues

1. Sale Stage May Stall

The `TokenSale` contract implements logic involving the enforcement of minimum and maximum participation levels, the process of returning change under certain purchase conditions and token crowdsale fundraising caps. There is a logic bug at the intersection of these concerns in the `_preValidatePurchase` function. This bug prevents the early switching from one sale stage to the next `BREAK` or `FINALAIZED` as is supposed to happen when no more contributions can be made to the current stage.

This logic is shown in the code excerpt below:

```
232 uint256 acceptedValue = _weiAmount;
233 if(contributions[_beneficiary].add(acceptedValue) > maximumContribution){
234   changeDue = (contributions[_beneficiary].add(acceptedValue)).sub(maximumContribution);
235   acceptedValue = acceptedValue.sub(changeDue);
236 }
237 uint256 currentCap = getCurrentCap();
238 uint256 raised = getRaisedForCurrentStage();
239 if(raised.add(acceptedValue) > currentCap){
240   changeDue = changeDue.add(raised.add(acceptedValue).sub(currentCap));
241   acceptedValue = _weiAmount.sub(changeDue);
242   capReached = true;
243 }
244 require(acceptedValue >= minimumContribution, "Contribution below minimum");
```

Consider the following situation:

- An incoming transaction arrives with a value less than the `maximumContribution` and equal to or larger than the `minimumContribution`.
- The difference between the current fundraising cap and the current fundraising amount is less than the `minimumContribution`. In other words, the current fundraising amount is very, very close to the cap.
- The current fundraising cap will require that change be made and the accepted value reduced.
- That change forces the accepted contribution to be below the `minimumContribution`.
- The last `require` statement will 'fail', reverting all the changes from that transaction.

This can happen when successive contributions have brought the amount of ether raised for the current stage to within less than `minimumContribution` amount from the current sale cap.

This can be solved by implementing the following modification to the second part of the `_preValidatePurchase` function as shown below:

```
237 uint256 currentCap = getCurrentCap();
238 uint256 raised = getRaisedForCurrentStage();
239 if(raised.add(acceptedValue) > currentCap.sub(minimumContribution)){
```

```
240   if(raised.add(acceptedValue) > currentCap){
241     changeDue = changeDue.add(raised.add(acceptedValue).sub(currentCap));
242     acceptedValue = _weiAmount.sub(changeDue);
243   }
244   capReached = true;
245 }
246 require(acceptedValue >= minimumContribution, "Contribution below minimum");
```

This will effectively set the `capReached` if no other contributions can be made to the current sale stage (because no contribution lower than `minimumContribution` can be made). This would also remove the need for the `updateStage` function to ever be called, allowing it to be entirely removed from the contract (the timed transition can still be forced through a direct call to the `saleOpen` function).

2. Inconsistent/Confusing Critical Constant Declaration and Behavior

Critical constants are those values which are necessary for a user to clearly understand the implementation and behavior prior to making a decision regarding participation. New Alchemy strongly believes that high clarity and transparency standards directly correspond to the trust and confidence users place in the contracts and their owners. As they stand, the contracts demonstrate three different implementations of critical constants and two different behaviors.

Some constants are effectively hardcoded into the source code during development (albeit in the constructor) and not subject to change. An example is shown below:

```
98 minimumContribution = 0.5 ether;
99 maximumContribution = 100 ether;
```


Some constants are passed as parameters to the constructor during deployment and may be changed on a per deployed-instance basis. An example is shown below:

```
92 constructor(uint256 _rate, address _wallet, ERC20 _token, uint256 _presaleCap,
93             uint256 _mainCap) public Crowdsale(_rate,_wallet,_token) {
94   presale_TokenCap = _presaleCap.div(rate).mul(1250); // .. (cap/rate*1000)*1.25
95   mainSale_TokenCap = _mainCap.div(rate).mul(1000);
96   presale_Cap = _presaleCap;
97   mainSale_Cap = _mainCap;
```

Some constants are buried within complex logic. While these are obviously placed during development and not subject to change, finding and understanding them may be challenging. Their definition is placed amongst complex code execution logic. An example is shown below:

```
183 function distributeTokens() public onlyOwner atStage(Stages.FINALAIZED) {
184   require(!distributed);
185   require(checkPercentages(40));  // Magic number -> Only 60 will be sold,
186                                   // therefore all other must be less than 40
187   uint256 totalTokens = (presale_TokesSold.add(mainSale_TokesSold)).mul(10).div(6);
188   for(uint i = 0; i < partners.length; i++){
189     uint256 amount = percentages[partners[i]].mul(totalTokens).div(1000);
190     _deliverTokens(partners[i], amount);
191   }
192   require(SolidToken(token).finishMinting());
193   distributed = true;
194 }
```

Some constants appear to be placeholders for later use as shown by the excerpt from `Distributable.sol` below. The source code comments seem to confirm this interpretation. The `percentages` contract state variable is never set or modified outside of the constructor (even within the whole package of contracts). The same is true for the `partners` address array.

```
9   bool distributed;
10  //Not actual addresses
11  address[] public partners = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x10];
12
13  mapping(address => uint) public percentages;
14
15  constructor(){
16    //Not actual percentages
17    percentages[0x01] = 10;
18    percentages[0x02] = 10;
19    percentages[0x03] = 10;
20    percentages[0x04] = 10;
21    percentages[0x05] = 10;
```

New Alchemy strongly recommends to clearly hardcode as many critical constants as possible directly into the contract as constant state variables with a distinctive convention (such as in SNAKE_CASE). This should be done at development time. This makes it easy for the user to find, understand and have confidence in.

Second, New Alchemy recommends to minimize the number of critical constants passed as parameters to the contract constructor and to clarify if/how these constants may change.

Third, eliminate 'magic numbers'. Declare them as constant state variables.

In addition to the above, OpenZeppelin functionality such as (changeable) ownership and pause capabilities should be documented. As it stands, the user needs to research this inherited functionality and determine if important aspects are overridden.

Minor Issues

3. Two Functions Missing `onlyOwner`

Several functions can benefit from the `onlyOwner` modifier. Specifically,

- The `enableTransfer()` function on line 32 of `SolidToken.sol`.
- The `updateStage()` function on line 130 of `TokenSale.sol`.

While this doesn't create a security concern as these functions are only supposed to help trigger timing based functionality (at a pre-determined point in time), changing the contract internal state is usually only done by the contract owner.

4. Unnecessary Loss of Precision

When division is done prior to multiplication there is an unnecessary loss of precision in the result.

The constructor for `TokenSale` is shown below (with adapted formatting). There are two examples of division prior to multiplication resulting in an unnecessary loss of precision.

```
constructor(uint256 _rate, address _wallet, ERC20 _token, uint256 _presaleCap,
            uint256 _mainCap) public Crowdsale(_rate,_wallet,_token) {
  presale_TokenCap = _presaleCap.div(rate).mul(1250);  // <removed for space>
  mainSale_TokenCap = _mainCap.div(rate).mul(1000);
  presale_Cap = _presaleCap;
  mainSale_Cap = _mainCap;

  minimumContribution = 0.5 ether;
  maximumContribution = 100 ether;
}
```


The loss of precision can may accumulate and become significant when a large number of sequential operations are performed and/or when the magnitude of operands is similar and small. It is believed that neither condition applies in this instance, hence a minor issue.

Since multiplication and division are associative operations their ordering is theoretically unimportant. However, in practice performing the multiplication prior to division will result in less loss of precision.

5. Pause Functionality Incorporated but not Used

The `TokenSale.sol` contract imports `Pausable` functionality from OpenZeppelin and inherits it on the contract declaration. However, the contract never makes use of this functionality. Specifically, `Pausable` provides function modifiers named `whenPaused()` and `whenNotPaused()` that are intended to block and unblock activity. Pausable capabilities are not used within the additional functionality

imported from OpenZeppelin. The `TokenSale.sol` contract currently has extensive sale-stage management functionality that may present challenges to make interwork correctly.

6. Lack of Short-Address Attack Protections

Some Ethereum clients may create malformed messages if a user is persuaded to call a function on a contract with an address that is not a full 20 bytes long. In such a "short-address attack", an attacker generates an address whose last byte is 0x00, then sends the first 19 bytes of that address to a victim. When the victim makes a contract function call, it appends the 19-byte address to `msg.data` followed by a value. Since the high-order byte of the value is almost certainly 0x00, reading 20 bytes from the expected location of the address in `msg.data` will result in the correct address. However, the value is then left-shifted by one byte, effectively multiplying it by 256 and potentially causing the victim to transfer a much larger number of tokens than intended. `msg.data` will be one byte shorter than expected, but due to how the EVM works, reads past its end will just return 0x00.

This attack affects functions that transfer tokens to destination addresses, where the function parameters include a destination address followed immediately by a value. In the Solidified contracts, such functions include:

- `SolidToken.sol:57 function transfer(address _to, uint256 _value)`
- `SolidToken.sol:70 function transferFrom(address _from, address _to, uint256 _value)`
- `SolidToken.sol:46 function mint(address _to, uint256 _amount)`

While the root cause of this flaw is buggy serializers and how the EVM works, it can be easily mitigated in contracts. When called externally, an affected function should verify that `msg.data.length` is *at least* the minimum length of the function's expected arguments (for instance, `msg.data.length` for an external call to `transfer` should be at least 68: 4 for the hash, 32 for the address (including 12 bytes of padding), and 32 for the value; some clients may add additional padding to the end). This can be implemented in a modifier. External calls can be detected in the following ways:

- Compare the first four bytes of `msg.data` against the function hash. If they don't match, then the call is internal and no short-address check is necessary.
- Avoid creating `public` functions that may be subject to short-address attacks; instead create only `external` functions that check for short addresses as described above. `Public` functions can be simulated by having the external functions call `private` or `internal` functions that perform the actual operations and that do not check for short-address attacks.

Whether or not it is appropriate for contracts to mitigate the short-address attack is a contentious issue among smart-contract developers. Many, including those behind the OpenZeppelin project, have explicitly chosen not to do so. While it is New Alchemy's position that there is value in protecting users by incorporating low-cost mitigations into likely target functions, Solidified would not stand out from the community if they also choose not to do so.

7. Lack of two-phase ownership transfer

In contracts that inherit the common `Ownable` contract from the OpenZeppelin project[3], a contract has a single owner. That owner can unilaterally transfer ownership to a different address. However, if the owner of a contract makes a mistake in entering the address of an intended new owner, then the contract can become irrecoverably unowned.

In order to preclude this, New Alchemy recommends implementing a two-phase ownership transfer. In this model, the original owner designates a new owner, but does not actually transfer ownership. The new owner then accepts ownership and completes the transfer. This can be implemented as follows:

```
contract Ownable {
    address public owner;
    address public newOwner

    event OwnershipTransferred(address indexed oldOwner, address indexed newOwner);

    function Ownable() public {
        owner = msg.sender;
        newOwner = address(0);
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address _newOwner) public onlyOwner {
        require(address(0) != _newOwner);
        newOwner = _newOwner;
    }

    function acceptOwnership() public {
        require(msg.sender == newOwner);
        OwnershipTransferred(owner, msg.sender);
        owner = msg.sender;
        newOwner = address(0);
    }
}
```

[3] https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/ownership/Ownable.sol

8. ERC20 double-spend attack

The standard ERC20 interface, implemented in `ERC20Token`, has a design flaw: if some user Alice wants to change the allowance that it grants to some other user Bob, then Alice can only check if Bob has already spent its allowance before issuing the transaction to change Bob's allowance. However, Bob can still spend the original allowance before the transaction changing the allowance is mined, allowing Bob to spend both the pre-change and post-change allowances[4]. In order to have a high probability of successfully spending the pre-change allowance after the victim has verified that it is not yet spent, it may be necessary for the attacker to wait until the transaction to change the allowance is issued, then issue a spend transaction with an unusually high gas price to ensure that the spend transaction is mined before the allowance change. More detail on this flaw are available at https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/ and https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729.

Due to this flaw, the safe use of the ERC20 `approve` interface to change allowances requires that: 1. allowances only change between zero and non-zero 2. allowances change at most once in a block

These restrictions allow Alice to safely change Bob's allowance by first setting it to zero, waiting for that transaction to be mined, verifying that Bob didn't spend its original allowance, then finally setting the allowance to the new value. Requiring this sequence of operations in `approve` would violate the ERC20 standard, though users can still take this approach even without changes in the `approve` implementation.

To make it more convenient to change allowances, New Alchemy recommends providing `increaseApproval` and `decreaseApproval` functions that add or subtract to the existing allowances rather than overwriting them. This effectively moves the check of whether Bob has spent its allowance to the time that the transaction is mined, removing Bob's ability to double-spend. Clients that are aware of this non-standard interface can use it rather than `approve`; using `approve` remains open to abuse. This is approach is implemented by the current version of the OpenZeppelin library[5].

Alternatively, only allow `approve` to change allowances between zero and non-zero and don't allow multiple changes to a user's allowance in the same block. In order to change Bob's allowance, Alice must first set it to zero, wait until that transaction is mined, verify that the original allowance was not spent, then finally set the allowance to the new value. Requiring this sequence of operations by implementing restrictions in `approve` would violate the ERC20 standard, though users can still take this approach even without changes in the `approve` implementation.

Since both approaches are outside of the ERC20 standard, both approaches require user cooperation to work properly. Accordingly, Solidified should provide documentation advising its users on how they should manage other users' allowances.

[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#approve
[5] https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

Line by line comments

This section lists comments on design decisions and code quality made by New Alchemy during the review. They are not known to represent security flaws.

Source file `SolidToken.sol`

Line 3:

Double quotes are preferred over single quote for import statements

Line 9:

Contract constant state variable `decimals` is unused

Lines 11-12:

Specify visibility of contract constant state variables `DECIMAL_PLACES` and `supplyCap` (defaults to `public`)

Line 12:

A distinctive convention, such as SNAKE_CASE, is preferred for constants

Source file `Distributable.sol`

Lines 9, 15:

Specify visibility of contract state variable `distributed` and `constructor()` function (defaults to `public`)

Line 11, 25, 26, 32

The last two entries of `partners` and `percentages` are the hexadecimal constants `0x09` and `0x10` which represent a discontinuity among the preceding strictly-incremental values. It appears the loop on line 32 functions correctly despite this. However, this may be unintuitive and brittle logic which can be a future source of bugs. Presumably these are placeholders that will be replaced.

Line 13:

Convert mapping value `uint` to `uint256` to improve clarity

Line 29:

Replace deprecated `constant` modifier with `view` on checkPercentages() function

Pervasive

This source file has very minimal comments which may obscure its intent.

Source file `TokenSale.sol`

Lines 3-7:

Double quotes are preferred over single quote for import statements

Line 13:

The contract state variable `currentStage` is never explicitly initialized.

Lines 35-36:

Specify visibility of contract state variables `changeDue` and `capReached` (defaults to `public`)

Lines 93-94:

Multiply before divide to maintain maximum precision

Lines 98-99:

Move `minimumContribution` and `maximumContribution` into contract constant state variables

Lines 130:

Specify visibility of `updateStage()` function (defaults to `public`)

Lines 44, 138, 176, 183, 215:

The enumerated value `FINALAIZED` is consistently misspelled

Line 229: Require Statement Attempting to Change Sale Stage

The `require` statement calls a function called `saleOpen` which in turn calls the `timedTransition` modifier which can change the stage of the sale if the timing requires it to. However if the sale is changed to a stage where the `saleOpen` function would return `false` (any stage other than `PRESALE` and `MAINSALE`), that state change will be reverted back as the `require` will make the transaction fail. This could be prevented by not failing the transaction and instead transferring back the ether to the `_beneficiary`.

Pervasive:

Move 'magic numbers' into contract constant state variables. One example from `TokenSale.sol` implements the 20% discount stated in the `README.md` and requires the user to realize that 1/0.80 = 1.25.

```
amount = amount.add(amount.mul(25).div(100)); //Add bonus
```