Hello Gold GBT Audit

Zero Knowledge Labs Auditing Servives

AUTHOR: MATTHEW DI FERRANTE

2017-01-11

Audited Material Summary

The audit consists of the following contracts:

```
a7aea536805bccb5fe80a7f1f2d3fd07ae22684cbb26e995fed8c2ccaf06b809  GoldBackedToken2.sol
d963022a60415010a1628fde86d84f7e107b2211206d50756a5543056cee7adc  GoldFees.sol
```

Security

SafeMath is not used in `GoldFees`, and the coding style of `GoldBackedToken` is more complex than it needs to be.

The function `addAllocationPartOne`, and functions that make use of `calcFees` and general fee calculation paths specifically could benefit from being rewritten in cleaner style.

SafeMath is not used in the entirety of `GoldBackedContract`. The following functions need additional SafeMath:

`totalSupply` `updateBalance` `addAllocationPartOne`

The `addAllocation` functions and the general allocation maths is confusing and would benefit from a more structured approact.

Though the audit found no critical issues, there may be edge cases in the fee calculation arithmetic that cannot be caught by SafeMath. A formalized algorithmic description of expected fee behaviour would be helpful so that users can ensure that the functionality matches the intent.

GoldBackedToken2.sol

Reclaimable

reclaim

```
    function reclaim(ERC20Basic token)
        public
        onlyOwner
    {
        address reclaimer = msg.sender;
        if (token == RECLAIM_ETHER) {
```

```
            reclaimer.transfer(this.balance);
        } else {
            uint256 balance = token.balanceOf(this);
            require(token.transfer(reclaimer, balance));
        }
    }
```

The `Reclaimable` contract implements the `reclaim` function which allows the contract owner to recover ether and tokens that are mistakenly sent to this address.

GBTBasic

The `GBTBasic` contract is a convencience interface implementation used for the migration process. It holds only two callable functions, both constant.

currentAllocationLength

```
    function currentAllocationLength() view public returns (uint256) {
        return currentAllocations.length;
    }
```

`currentAllocationLength` returns the length of the `currentAllocations` array.

aotLength

```
    function aotLength() view public returns (uint256) {
        return allocationsOverTime.length;
    }
```

`aotLength` returns the length of the `allocationsOverTime` array

GoldBackedToken

The `GoldBackedToken` contract implements the upgraded replacement token to `gbt.thetoken.eth`.

It is an ERC20 token, with `Pausable`, `Ownable`, and `Reclaimable` functionality.

```
contract GoldBackedToken is Ownable, ERC20, Pausable, GBTBasic,
    Reclaimable
```

Security

The contract has no critical vulnerabilities.

Constructor

```
    function GoldBackedToken(GoldFees feeCalc, GBTBasic _oldToken) public
        {
        uint delta = 3799997201200178500814753;
        feeCalculator = feeCalc;
        oldToken = _oldToken;
        // now migrate the non balance stuff
        uint x;
        for (x = 0; x < oldToken.aotLength(); x++) {
            Allocation memory al;
            (al.amount, al.date) = oldToken.allocationsOverTime(x);
            allocationsOverTime.push(al);
        }
        allocationsOverTime[3].amount = allocationsOverTime[3].amount.sub(
            delta);
        for (x = 0; x < oldToken.currentAllocationLength(); x++) {
            (al.amount, al.date) = oldToken.currentAllocations(x);
            al.amount = al.amount.sub(delta);
            currentAllocations.push(al);
        }

        // 1st Minting : TxHash 0
            x8ba9175d77ed5d3bbf0ddb3666df496d3789da5aa41e46228df91357d9eae8bd

        // amount = 528359800000000000000;
        // date = 1512646845;

        // 2nd Minting : TxHash 0
            xb3ec483dc8cf7dbbe29f4b86bd371702dd0fdaccd91d1b2d57d5e9a18b23d022

        // date = 1513855345;
        // amount = 1003203581831868623088;
```

```
26
27          // Get values of first minting at second minting date
28          // feeCalc(1512646845,1513855345,528359800000000000000) =>
                (527954627221032516031,405172778967483969)
29
30          mintedGBT.date = 1515700247;
31          mintedGBT.amount = 15293134908616925416 44;
32      }
```

The constructor for `GoldBackedToken` sets the fees and the old token's address, and migrates the old token's allocations to the new contract.

It also sets a new minted `GBT` amount and date.

totalSupply

```
1   function totalSupply() view public returns (uint256) {
2       uint256 minted;
3       uint256 mFees;
4       uint256 uminted;
5       uint256 umFees;
6       uint256 allocated;
7       uint256 aFees;
8       (minted,mFees) = calcFees(mintedGBT.date,now,mintedGBT.amount);
9       (uminted,umFees) = calcFees(unmintedGBT.date,now,unmintedGBT.amount)
            ;
10      (allocated,aFees) = calcFees(currentAllocations[0].date,now,
            currentAllocations[0].amount);
11      if (minted+allocated>uminted) {
12        return minted + allocated - uminted;
13      } else {
14        return 0;
15      }
16  }
```

The `totalSupply` function returns the amount of tokens minted and allocated, minus the unminted tokens. The values are derived from `unmintedGBT`, `mintedGBT`, and `currentAllocations`, after being passed through `calcFees`.

updateMaxAllocation

```
    function updateMaxAllocation(uint256 newMax) public onlyOwner {
        require(newMax > 38 * 10**5 * 10**decimals);
        maxAllocation = newMax;
    }
```

The `updateMaxAllocation` function allows the contract owner to change the reward allocation limit.

setFeeCalculator

```
    function setFeeCalculator(GoldFees newFC) public onlyOwner {
        feeCalculator = newFC;
    }
```

The `setFeeCalculator` function allows the owner to set a new fee calculator contract address.

calcFees

```
    function calcFees(uint256 from, uint256 to, uint256 amount) view
       public returns (uint256 val, uint256 fee) {
        return feeCalculator.calcFees(from,to,amount);
    }
```

The `calcFees` function returns the value and fees from the `feeCalculator`.

migrateBalance

```
    function migrateBalance(address where) public {
        if (!updated[where]) {
            uint256 am;
            uint256 lu;
            uint256 ne;
            uint256 al;
            (am,lu,ne,al) = oldToken.balances(where);
            balances[where] = Balance(am,lu,ne,al);
            updated[where] = true;
        }

    }
```

The `migrateBalance` function allows a user to migrate the balance from their old token contract to the new contract. It can only be called once per address.

update

```
    function update(address where) internal {
        uint256 pos;
        uint256 fees;
        uint256 val;
        migrateBalance(where);
        (val,fees,pos) = updatedBalance(where);
        balances[where].nextAllocationIndex = pos;
        balances[where].amount = val;
        balances[where].lastUpdated = now;
    }
```

The `update` function is an internal function that attempts to migrate balance for the address it is called on, and sets `nextAllocation`, `amount` and `lastUpdated` based on the return of `updatedBalance`.

updatedBalance

```
    function updatedBalance(address where) view public returns (uint val,
       uint fees, uint pos) {
        uint256 cVal;
        uint256 cFees;
        uint256 cAmount;

        uint256 am;
        uint256 lu;
        uint256 ne;
        uint256 al;
        Balance memory bb;

        // calculate update of balance in account
        if (updated[where]) {
            bb = balances[where];
            am = bb.amount;
            lu = bb.lastUpdated;
            ne = bb.nextAllocationIndex;
            al = bb.allocationShare;
```

```
        } else {
            (am,lu,ne,al) = oldToken.balances(where);
        }
        (val,fees) = calcFees(lu,now,am);
        // calculate update based on accrued disbursals
        pos = ne;
        if ((pos < currentAllocations.length) && (al != 0)) {
            cAmount = currentAllocations[ne].amount * al / allocationPool;
            (cVal,cFees) = calcFees(currentAllocations[ne].date,now,
                cAmount);
        }
        val = val.add(cVal);
        fees = fees.add(cFees);
        pos = currentAllocations.length;
    }
```

The `updatedBalance` function returns the amount, fees and allocations position for an address, after applying fees for current and accrued disbursals.

Security note: This function needs to use SafeMath ofr the accrued disbursals

balanceOf

```
    function balanceOf(address where) view public returns (uint256 val) {
        uint256 fees;
        uint256 pos;
        (val,fees,pos) = updatedBalance(where);
        return ;
    }
```

ERC20 balanceOf, returning the value from `updatedBalance`.

partAllocationLength

```
    function partAllocationLength() view public returns (uint) {
        return partAllocations.length;
    }
```

Returns the length of the `partAllocattions` array.

addAllocationPartOne

```
    function addAllocationPartOne(uint newAllocation,uint numSteps)
        public
        onlyMinter
    {
        require(partPos == 0);
        uint256 thisAllocation = newAllocation;

        require(totAllocation < maxAllocation);     // cannot allocate
           more than this;

        if (currentAllocations.length > partAllocations.length) {
            partAllocations = currentAllocations;
        }

        if (totAllocation + thisAllocation > maxAllocation) {
            thisAllocation = maxAllocation - totAllocation;
            log0("max alloc reached");
        }
        totAllocation = totAllocation.add(thisAllocation);

        GoldAllocation(thisAllocation,now);

        Allocation memory newDiv;
        newDiv.amount = thisAllocation;
        newDiv.date = now;
        // store into history
        allocationsOverTime.push(newDiv);
        // add this record to the end of currentAllocations
        partL = partAllocations.push(newDiv);
        // update all other records with calcs from last record
        if (partAllocations.length < 2) { // no fees to consider
            PartComplete();
            currentAllocations = partAllocations;
            FeeOnAllocation(0,now);
            return;
        }
        //
        // The only fees that need to be collected are the fees on
           location zero.
```

```
38          // Since they are the last calculated = they come out with the
                break
39          //
40          for (partPos = partAllocations.length - 2; partPos >= 0; partPos--
                ) {
41              (partAllocations[partPos].amount,partFees) = calcFees(
                    partAllocations[partPos].date,now,partAllocations[partPos].
                    amount);
42
43              partAllocations[partPos].amount = partAllocations[partPos].
                    amount.add(partAllocations[partL - 1].amount);
44              partAllocations[partPos].date = now;
45              if ((partPos == 0) || (partPos == partAllocations.length-
                    numSteps)) {
46                  break;
47              }
48          }
49          if (partPos != 0) {
50              StillToGo(partPos);
51              return; // not done yet
52          }
53          PartComplete();
54          FeeOnAllocation(partFees,now);
55          currentAllocations = partAllocations;
56      }
```

This function allows the minter to add a new allocation to `currentAllocations`, and the amount of allocations affected by the call can be limited by `numSteps`.

Security note: Perhaps pause the token while `partPos` is not 0 to prevent the contract from ending up in an inconsistent state

addAllocationPartTwo

```
1       function addAllocationPartTwo(uint numSteps)
2           public
3           onlyMinter
4       {
5           require(numSteps > 0);
6           require(partPos > 0);
7           for (uint i = 0; i < numSteps; i++ ) {
8               partPos--;
```

```
            (partAllocations[partPos].amount,partFees) = calcFees(
               partAllocations[partPos].date,now,partAllocations[partPos].
               amount);
            partAllocations[partPos].amount = partAllocations[partPos].
               amount.add(partAllocations[partL - 1].amount);
            partAllocations[partPos].date = now;
            if (partPos == 0) {
                break;
            }
        }
        if (partPos != 0) {
            StillToGo(partPos);
            return; // not done yet
        }
        PartComplete();
        FeeOnAllocation(partFees,now);
        currentAllocations = partAllocations;
    }
```

The `addAllocationPartTwo` function allows continuation from `addAllocationPartOne` in case the latter was not able to complete due to gas limit considerations (when numSteps is too low).

setHGT

```
    function setHGT(address _hgt) public onlyOwner {
        HGT = _hgt;
    }
```

The `setHGT` function allows the contract owner to set the HGT address.

parentFees

```
    function parentFees(address where) public whenNotPaused {
        require(msg.sender == HGT);
        update(where);
    }
```

The `parentFees` function updates balances for an address. The caller of the function must be `HGT`, and can only be called when the token is not paused.

parentChange

```
    function parentChange(address where, uint newValue) public
        whenNotPaused { // called when HGT balance changes
        require(msg.sender == HGT);
        balances[where].allocationShare = newValue;
    }
```

The `parentChange` function allows the HGT address to set a new allocation share for an address. This function is only callable when the token is not paused.

transfer

```
    function transfer(address _to, uint256 _value) public whenNotPaused
        returns (bool ok) {
        require(_to != address(0));
        update(msg.sender);              // Do this to ensure sender has
            enough funds.
        update(_to);

        balances[msg.sender].amount = balances[msg.sender].amount.sub(
            _value);
        balances[_to].amount = balances[_to].amount.add(_value);
        Transfer(msg.sender, _to, _value); //Notify anyone listening that
            this transfer took place
        return true;
    }
```

Standard ERC20 transfer function, with additional `whenNotPaused` modifier and balance `update()` calls for the affected address.

transferFrom

```
    function transferFrom(address _from, address _to, uint _value) public
        whenNotPaused returns (bool success) {
        require(_to != address(0));
        var _allowance = allowance[_from][msg.sender];

        update(_from);              // Do this to ensure sender has enough
            funds.
```

```
6         update(_to);
7
8         balances[_to].amount = balances[_to].amount.add(_value);
9         balances[_from].amount = balances[_from].amount.sub(_value);
10        allowance[_from][msg.sender] = _allowance.sub(_value);
11        Transfer(_from, _to, _value);
12        return true;
13    }
```

Standard ERC20 transferFrom, with additional `whenNotPaused` modifier and balance `update()` calls for the affected addresses.

approve

```
1     function approve(address _spender, uint _value) public whenNotPaused
          returns (bool success) {
2         require((_value == 0) || (allowance[msg.sender][_spender] == 0));
3         allowance[msg.sender][_spender] = _value;
4         Approval(msg.sender, _spender, _value);
5         return true;
6     }
```

Standard ERC20 approve function, with added `whenNotPaused` modifier.

increaseApproval

```
1   function increaseApproval(address _spender, uint _addedValue) public
        returns (bool) {
2     allowance[msg.sender][_spender] = allowance[msg.sender][_spender].add(
          _addedValue);
3     Approval(msg.sender, _spender, allowance[msg.sender][_spender]);
4     return true;
5   }
```

`increaseApproval` allows a token holder to increase approved allowance for a spender.

decreaseApproval

```
  function decreaseApproval(address _spender, uint _subtractedValue)
     public returns (bool) {
    uint oldValue = allowance[msg.sender][_spender];
    if (_subtractedValue > oldValue) {
      allowance[msg.sender][_spender] = 0;
    } else {
      allowance[msg.sender][_spender] = oldValue.sub(_subtractedValue);
    }
    Approval(msg.sender, _spender, allowance[msg.sender][_spender]);
    return true;
  }
```

`decreaseApproval` allows a token holder to decrease approved allowance for a spender.

allowance

```
    function allowance(address _owner, address _spender) public view
       returns (uint remaining) {
        return allowance[_owner][_spender];
    }
```

Standard ERC20 `allowance` function.

setMinter

```
    function setMinter(address minter) public onlyOwner {
        authorisedMinter = minter;
    }
```

The `setMinter` function allows the contract owner to set a new minter address.

mintTokens

```
    function mintTokens(address destination, uint256 amount)
        onlyMinter
        public
    {
        require(msg.sender == authorisedMinter);
```

```
6           update(destination);
7           balances[destination].amount = balances[destination].amount.add(
                amount);
8           TokenMinted(destination,amount);
9           Transfer(0x0,destination,amount); // ERC20 compliance
10          //
11          // TotalAllocation stuff
12          //
13          uint256 fees;
14          (mintedGBT.amount,fees) = calcFees(mintedGBT.date,now,mintedGBT.
                amount);
15          mintedGBT.amount = mintedGBT.amount.add(amount);
16          mintedGBT.date = now;
17      }
```

The `mintTokens` function allows the minter to allocate new tokens to an address.

It performs an update/migration for the address, and adds new tokens to it, emitting a `Transfer` event from `0x00` to the address.

`mintedGBT.amount` and `mintedGBT.date` are updated accordingly.

burnTokens

```
1       function burnTokens(address source, uint256 amount)
2           onlyMinter
3           public
4       {
5           update(source);
6           balances[source].amount = balances[source].amount.sub(amount);
7           TokenBurned(source,amount);
8           Transfer(source,0x0,amount); // ERC20 compliance
9           //
10          // TotalAllocation stuff
11          //
12          uint256 fees;
13          (unmintedGBT.amount,fees) = calcFees(unmintedGBT.date,now,
                unmintedGBT.amount);
14          unmintedGBT.date = now;
15          unmintedGBT.amount = unmintedGBT.amount.add(amount);
16      }
```

The `burnTokens` function allows the minter to destroy tokens for an address.

It migrates the balance of the address, if it has not already been migrated, and subtracts the amount to be burned from the address' token balance.

On success, it emits a `Transfer` event that signals a move to the `0x0` address, and the `unmintedGBT.date` and `unmintedGBT.amount` variables are updated.

Disclaimer

This audit concerns only the correctness of the Smart Contracts listed, and is not to be taken as an endorsement of the platform, team, or company.

Audit Attestation

This audit has been signed by the key provided on https://keybase.io/mattdf - and the signature is available on https://github.com/mattdf/audits/