MCoin Token and Crowdsale Audit Report

May 21st, 2018

Audited Material Summary

This audit report consists of the crowdsale distribution contract `MCoinDistribution.sol`, the tokens MCoin (`MinableCoin.sol`) and M5Coin (`MinableM5Token.sol`) and related files. Analysis of tokens is based on the available mocks. Commit hash under which the audit was performed is: `aee6e4277bbce0e2b71ae45acbbcd90b03ae5197`, from Apr. 1st, 2018.

Audit Result Summary

Positive impressions:

- Overall, no high-risk vulnerabilities, i.e., that are exploitable by some malicious external users, were found.
- The intended behavior of the main MCoin contract is clear.
- The testing suite is exhaustive and impressive given the complexity of the model, having many different actors.
- See the disclaimer in the epilogue of this audit report for information regarding what aspects of the MCoin ecosystem were not audited.

Concerns:

Our comments pertain mainly to issues of trust, confidence, reliability and clarity. Some of those stem from inherent design flaws in the EVM. Others from confusing semantics of the tokens, especially the M5 token. Main concerns are:

- Upgradeable contracts such as M5 Token and M5 Logic. See the relevant section for details.
- External actor roles such as *GDP Oracle*, *Upgrade Manager, Distribution contract owner* and *Foundation wallet*. These roles pose two kinds of risks:
    1) Centralization. This requires separate review of the mechanisms with which external roles are implemented, and also entails real-world guarantees regarding the responsibility and accountability of these actors.
       - As a side note, all actors have their power confined within clear boundaries. No hidden abilities given to these external actors. We highlighted such abilities and the expected impact. In some cases, external actors roles are transient (e.g. distribution contract owner), diminishing the impact of changes. It should be considered if complete removal of these roles, once irrelevant, is beneficiary.
    2) Verification of intended behavior. Implementing mechanisms that reduce the risk for bounding non-compatible contracts to the MCoin ecosystem. Such mechanisms could be more elaborate APIs for ownership transferal, complemented with formally laid-out upgrade process, which includes end-to-end testing of the upgraded component in testnets first.
- M5 token's non-finalized behavior and specification. Every upgrade of the M5 tokens should be carefully planned and executed. At least some subset of it should have clear semantics already. The upgrade process is very risky due to the fact there are two upgradeable contracts, which are still tightly coupled to each other, as can be seen from some of the mocks.
  Importantly, this concern is greatly diminished in comparison to other common upgrade

techniques: Proxy contracts and dispatchers based on fallback functions.
The upgrade mechanism was clearly designed to isolate as much as possible the affected behavior. Transactions that flow through non-upgradeable functions will remain safe.

- The team should make sure to upgrade Solidity version to the latest and re-run all tests.
- Better documentation inside the contracts, and a more illustrative document or presentation that explains the essentials of the MCoin ecosystem. This includes explanation about the integration with the compliance store, and what to expect when M5 tokens are minted. (Also, there are some inconsistencies within the whitepaper, such as the table of contents.)

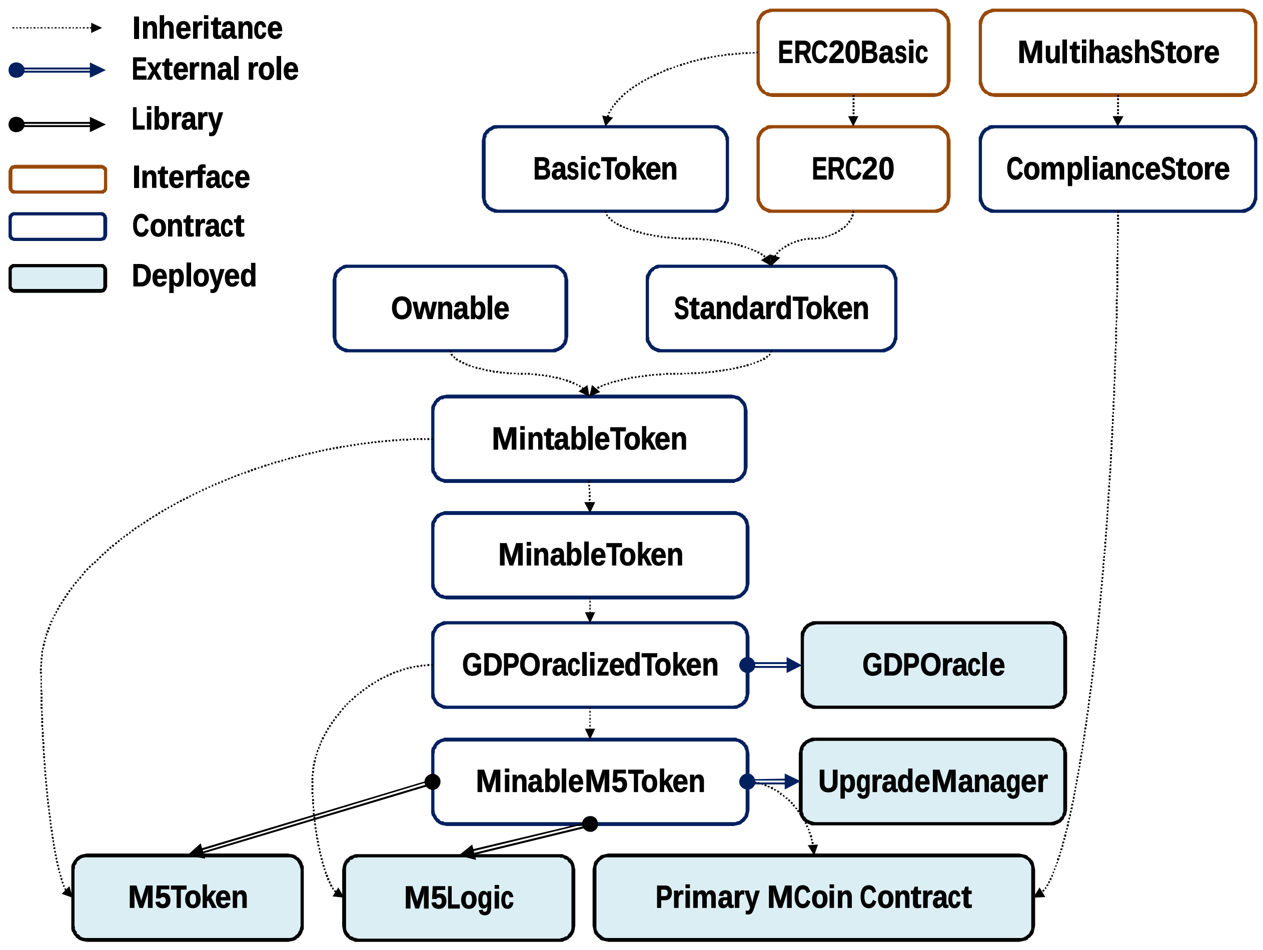
MCoin - Overview

MCoin is a GDP-regulated asset that complies with the ERC20 standard. In the most basic level, it allows token holders to use it as any other ERC20 token. But in addition, it can be committed for "mining". A token that is committed may be later withdrawn with an additional reward determined by the GDP oracle. This means MCoin's committed supply ("stake") is regulated. The possible issue of negative GDP is handled with special M5 tokens which are also tradable and compatible with the ERC20 standard. M5 tokens can be swapped for MCoin tokens once the GDP is back to positive. For more thorough review of this model please confer the whitepaper.

Below we show a diagram of illustrating the relation of MCoin contracts, interfaces, and libraries.

The first section overviews a simple view of the MCoin contract as an instance of `MinableToken` contract. The `MinableToken` contract has standard extended ERC20 functionality with minting, used during the crowdsale. `MinableToken` is implementing proof-of-stake mining. It allows locking a certain amount of MCoins as a form of investment. The locked MCoins' can generate reward depending on external factors – in this case, the GDP. The total amount of locked MCoins is called the total stake. MCoins that are not locked, but used in commerce, do not generate reward. We inspect the interaction between the different inheritance levels to see that representation invariants of each are retained.

We separate the analysis of the core logic to four subsections. The following section discusses only MCoin logic. The next section discusses the GDP Oracle. The third section addresses M5 logic and associated complications and risks. Finally, we provide a short review of the compliance store.



1. MCoin contract

Representation invariants:

- `Sum_a(balances[a]) + totalStake = totalSupply_`
  It should be noted that the original ERC20 invariant is: `Sum_a(balances[a]) = totalSupply_`
- After distribution has completed, `mintingFinished = True`
- `Sum_a(miners[a].value) = totalStake_`
- `Forall a. miners[a].value <= miners[a].atStake`
- `Forall a. miners[a].onBlockNumber <= currentBlockNumber`
- After distribution has completed, `owner` should have no additional roles in the contract

Immutable after construction: `upgradeManager`.

Many of these invariants are highly sensitive to the logic of M5, and should be rechecked upon every upgrade.

Non-view functions

Constructor (based on MinableM5TokenIntegrationMock):

```
function MinableM5TokenIntegrationMock(
                        address initialAccount,
                        uint256 initialSupply,
                        int256 blockReward,
                        address GDPOracle,
                        address upgradeManager
) public {
       require(0 < initialSupply);
       require(0 < blockReward);

       totalSupply_ = initialSupply;

       balances[initialAccount] = initialSupply;
       Transfer(0x0, initialAccount, initialSupply);
       blockReward_ = blockReward;
       BlockRewardChanged(0, blockReward_);

       GDPOracle_ = GDPOracle;
       GDPOracleTransferred(0x0, GDPOracle_);

       //M5 specific:
       M5Token_ = address(0);
       M5Logic_ = address(0);
       upgradeManager_ = upgradeManager;
}
```

Comments

1. The MCoin parent constructor is not yet fully specified. Analysis was performed based on one of the mocks. Integration audit must take into account the final instances of libraries `M5Token_` and `M5Logic_`, and external roles `upgradeManager_` and `GDPOracle_`.

Commit:

```
/**
* @dev commit amount for minning
* if user previously commited, add to an existing commitment.
* this is done by calling withdraw()
* then commit back previous commit + reward + new commit
* @param _value The amount to be commited.
* @return the commit value
* _value or prevCommit + reward + _value
*/
function commit(uint256 _value) public returns (uint256 commitValue) {
        require(0 < _value);
        require(_value <= balances[msg.sender]);
        commitValue = _value;
        uint256 prevCommit = miners[msg.sender].value;
        //In case user already commited, withdraw and recommit
        // new commitment value: prevCommit + reward + _value
        if (0 < prevCommit) {
                // Will revert if reward is negative
                commitValue = prevCommit.add(getReward(msg.sender)).add(commitValue);
                withdraw();
        }

        // sub will throw if there is not enough balance.
        balances[msg.sender] = balances[msg.sender].sub(commitValue);
        Transfer(msg.sender, address(0), commitValue);

        totalStake_ = totalStake_.add(commitValue);

        miners[msg.sender] = Commitment(
                commitValue, // Commitment.value
                block.number, // onBlockNumber
                totalStake_, // atStake = current stake + commitments value
                blockReward_ // onBlockReward
        );
        Commit(msg.sender, commitValue, totalStake_, blockReward_); // solium-disable-line

        return commitValue;
}
```

The method commits a non-negative amount of MCoins. The commit is represented using an entry in the `miners` mapping.

In the simple case, there is no previous commit. Thus, a new `Commitment` entry is created. The balance of the sender is decreased, and the same amount is transferred to the total stake. `Commitment` is a four-tuple

consisting of the commitment value, the current block number, the total stake, and the current block reward for the block of the commit.
A more complex case is when the user wishes to add `_value` to the current commitment. This is implemented by first withdrawing the current commitment, and then setting the sum of the previous commit, the reward, and the current argument `_value` as the new commit value. The commitment of the user is updated accordingly. If the reward is negative, the commit fails. A user will have to withdraw M5 tokens manually first.
The `totalSupply` of MCoins does not change in any of the cases.

Comments

2. Same comment for `withdraw`: What is the meaning of logging a transfer to address 0? Can it generate exceptions in ERC20-aware Dapps which assume that the 0 address may not have any transfers? Was "transfer-to-self" logging considered as an alternative?
3. It is worth documenting that the standard semantics of an ERC20 token change a bit in `MinableToken`. The total supply is equal to the sum of balances of all users *and* the total stake.
4. Calling `getReward` incurs some gas costs due to reading from the storage. The code can use the values returned from `withdraw` in order to calculate the new `commitValue`.
5. If there is a previous commitment, `withdraw()` is called. This means such a transaction generates in total 4 events instead of 2: `Transfer`, `Withdraw`, `Transfer`, `Commit` instead of just `Transfer+Commit`. Associated Dapps may have to write logic that handles specifically the combination of `Withdraw+Commit` in one transaction.
6. Put `getReward(msg.sender)` in a separate variable such as '`prevCommitReward`'. Add an assertion that `(prevCommitReward, prevCommit) = (reward, commitmentValue)` as returned from `withdraw()`. This will increase the confidence in the correctness of `commitValue`.

Withdraw:

```
/**
* @dev withdraw reward
* @return reward to withdraw
*/
function withdraw() public returns (uint256 reward, uint256 commitmentValue) {
       require(miners[msg.sender].value > 0);
       //will revert if reward is negative:
       reward = getReward(msg.sender);
       Commitment storage commitment = miners[msg.sender];
       commitmentValue = commitment.value;
       totalStake_ = totalStake_.sub(commitmentValue);
       totalSupply_ = totalSupply_.add(reward);
       balances[msg.sender] = balances[msg.sender].add(commitmentValue.add(reward));
       Transfer(address(0), msg.sender, commitmentValue.add(reward));
       commitment.value = 0;
       Withdraw(msg.sender, reward, commitmentValue);
       return (reward, commitmentValue);
}
```
This function withdraws the stake of the sender, including the GDP-based reward. If the calculated reward is negative, the transaction fails.

Comments

7. Consider extracting `commitmentValue.add(reward)` to a separate variable, e.g. a variable named `withdrawnSum`.
8. Make sure that the wallet checks first if the `withdraw()` function is expected to succeed by checking the reward prior to sending the transaction.
9. Consider deleting the entry from the `miners` map completely. This will help lower gas costs.

View functions

`commitmentOf`, `totalStake` and `blockReward` are straightforward. Additional useful view functions that could be added are for the other fields of the commitment entry, and a boolean view function checking if there is a commitment.

getReward:

```
/**
* @dev Calculate the reward if withdraw() happans on this block
* @return An uint256 representing the reward amount
*/
function getReward(address _miner) public view returns (uint256) {
       if (miners[_miner].value == 0) {
              return 0;
       }

       Commitment storage commitment = miners[_miner];

       int averageBlockReward = signedAverage(commitment.onBlockReward, blockReward_);
       require(0 <= averageBlockReward);
       uint256 effectiveBlockReward = uint(averageBlockReward);
       uint256 effectiveStake = average(commitment.atStake, totalStake_);
       uint256 numberOfBlocks = block.number.sub(commitment.onBlockNumber);

       uint256 miningReward = numberOfBlocks.mul(effectiveBlockReward)
                                                     .mul(commitment.value).div(effectiveStake);
       return miningReward;
}
```

The crux of MCoin token’s logic. This is a simplified version of the version that appears in the whitepaper. It computes the reward based on the average stake in the period between the commit block and the current block. Fluctuations in the value of the stake and in the reward are smoothed-out. Handling of average computation is careful and takes into account signed and unsigned numbers. Two auxiliary math functions, `average` and `signedAverage`, are also included.

Comments

10. Use uniform type names, `uint256` instead of `uint`, `int256` instead of `int` (these are aliases).

average, signedAverage:

```
/**
* @dev Calculate the average of two integer numbers
* 1.5 will be rounded down
* @return An uint256 representing integer average
*/
function average(uint a, uint b) public pure returns (uint) {
       return a.add(b).div(2);
}

/**
* @dev Calculate the average of two signed integers numbers
* 1.5 will be rounded down
* @return An int256 representing integer average
*/
function signedAverage(int256 a, int256 b) public pure returns (int256) {
       int ans = a + b;

       if (a > 0 && b > 0 && ans < 0)
       require(false);
       if (a < 0 && b < 0 && ans > 0)
       require(false);

       return ans / 2;
}
```

Auxiliary math functions for `getReward`.

Comments

11. Add a few more tests for both functions. Examples:
    - Signed average of negative and positive number resulting in a negative figure.
    - For `signedAverage`, add at least 2 examples where the sum of a and b is overflowing (both over- and under-flowing).
    - Overflow test for `average` (should pass thanks to `SafeMath`).
    - Examples with 0 for both `average` and `signedAverage`.

2. GDP Oracle token

The concrete implementation of the GDP oracle is currently not available for inspection. The GDP oracle is only capable of changing the values of the `blockReward_` and `GDPOracle_` fields. The `blockReward` can be set to either negative or positive using separate methods. Ownership of the oracle is similar to implementations of `Ownable`.

Non-view functions

All non-view functions are only executable by the GDP oracle.

transferGDPOracle:

```
/**
* @dev Allows the current GDPOracle to transfer control to a newOracle.
* @param newOracle The address to transfer ownership to.
*/
function transferGDPOracle(address newOracle) public onlyGDPOracle {
       require(newOracle != address(0));
       GDPOracleTransferred(GDPOracle_, newOracle);
       GDPOracle_ = newOracle;
}
```

This function is only executable by the GDP oracle. It has similar logic to `transferOwnership` of `Ownable`.

Comments

1. Due to the sensitiveness and importance of the GDP oracle, the transfer process should consist of two transactions: offering the role to some non-null address, and accepting the offer by the receiving GDP oracle, proving that it has the right API to manage the block reward and to change ownership. Another required method in such a protocol is to cancel an offer for a GDP oracle.

setPositiveGrowth:

```
/**
* @dev Chnage block reward according to GDP
* @param newBlockReward the new block reward in case of possible growth
*/
function setPossitiveGrowth(int256 newBlockReward) public onlyGDPOracle returns(bool) {
       // protect against error / overflow
       require(0 <= newBlockReward);
       BlockRewardChanged(blockReward_, newBlockReward);
       blockReward_ = newBlockReward;
}
```

For a non-negative number, updates the block reward and emits an event.

Comments

2. Fix typo in name of function: `setPositiveGrowth` instead of `setPos`**`s`**`itiveGrowth`

setNegativeGrowth:

```
/**
* @dev Chnage block reward according to GDP
* @param newBlockReward the new block reward in case of negative growth
*/
function setNegativeGrowth(int256 newBlockReward) public onlyGDPOracle returns(bool) {
        require(newBlockReward < 0);

        BlockRewardChanged(blockReward_, newBlockReward);
        blockReward_ = newBlockReward;
}
```

For a negative number, updates the block reward and emits an event.

View functions

GDPOracle():

```
/**
* @dev get GDPOracle
* @return the address of the GDPOracle
*/
function GDPOracle() public view returns (address) {
        return GDPOracle_;
}
```

This function returns the current address of the GDP oracle.

3. M5 Token Contract

M5 tokens are generated when the GDP is negative and the user wishes to withdraw MCoins, but can't. M5 tokens can be swapped for regular MCoins when the GDP becomes positive. M5 is a separate token contract with some different properties such as burnability. Its logic is upgradeable and is called using `delegatecall`, meaning it could have impact on the state of MCoin. In the sequel, we consider the non-upgradeable wrapper logic that calls the upgradeable contracts.

Non-view functions

withdrawM5:

```
/**
* @dev withdraw M5 reward, only appied to mining when GDP is negative
* @return reward
* @return commitmentValue
*/
function withdrawM5() public returns (uint256 reward, uint256 commitmentValue) {
        require(M5Logic_ != address(0));
        require(miners[msg.sender].value > 0);
        reward = getM5Reward(msg.sender);
        commitmentValue = miners[msg.sender].value;

        require(M5Logic_.delegatecall(bytes4(keccak256("withdrawM5()"))));
        return (reward,commitmentValue);
}
```

This function sets return values by reading from the state, and computes the reward with `getM5Reward` (which in turn also delegates), before executing `M5Logic` by `delegatecall`.

Comments

1. Consider checking if the block reward for the commitment is indeed negative. This logic could be implemented using a utility function in `MinableToken` that checks it like `getReward` does. This will reduce the mistake surface for `M5Logic`, which might mistakenly allow unwanted duplicity – being able to withdraw M5 tokens despite MCoins also being available for withdraw.
2. Consider using a more elaborate scheme for verifying that the values returned in the delegate call are in accordance with the values computed here.
3. Add an `M5Token` check in the requirements of the method, similarly to the one in swap.
4. Based on the example in `M5LogicMock3` used in "`fullExample`", it can be inferred that withdrawing M5 tokens results in getting back the original commitment in MCoins, plus a reward of M5 tokens, which is dependent on `getM5Reward`'s implementation. Is this behavior intended for all M5 token implementations? If the answer is yes, it should appear in the non-upgradable `MinableM5Token`. Otherwise, any upgrade will have to ensure equivalence up to the computation of the M5 reward and the swap reward.

swap:

```
/**
* @dev swap M5 tokens back to regular tokens when GDP is back to possitive
* @param _value The amount of M5 tokens to swap for regular tokens
* @return true
*/
function swap(uint256 _value) public returns (bool) {
        require(M5Logic_ != address(0));
        require(M5Token_ != address(0));

        require(M5Logic_.delegatecall(bytes4(keccak256("swap(uint256)")),_value));
        return true;
}
```

Thus function swaps M5 tokens with MCoins using a delegate call. The parameter is number of M5 tokens to swap.

Comments

5. It may be worth reading the returned amount of MCoins and checking it, or emitting a special own log, in case not implemented by the upgradeable contract.
6. In `M5LogicMock3` used in “`fullExample`”, there is no `Transfer` event for the balance increase in MCoin.

upgradeM5Token:

```
/**
* @dev Allows to set the M5 token contract
* @param newM5Token The address of the new contract
*/
function upgradeM5Token(address newM5Token) public onlyUpgradeManager {
        require(newM5Token != address(0));
        M5TokenUpgrade(M5Token_, newM5Token);
        M5Token_ = newM5Token;
}
```

This function updates the M5 token contract. It is only executable by the upgrade manager and if the upgrade capability was not disabled.

Comments

7. What happens with balances of the replaced M5 token contract?
   Does it mean there could be several copies of M5 tokens in the blockchain?
   How users will be able to swap them?
   Alternatively, is there a scheme for “copying” data between upgraded M5 token contracts, such that after the copying, old data is discarded?

upgradeM5Logic:

```
/**
* @dev Allows the upgrade the M5 logic contract
* @param newM5Logic The address of the new contract
*/
function upgradeM5Logic(address newM5Logic) public onlyUpgradeManager {
        require(newM5Logic != address(0));
        M5LogicUpgrade(M5Logic_, newM5Logic);
        M5Logic_ = newM5Logic;
}
```

Thus function updates the M5 logic contract. It is only executable by the upgrade manager and if the upgrade capability was not disabled.

Comments

8. As mentioned earlier, while upgradability is a very powerful property, it also creates enormous responsibility and risk with every update. As much as can be defined earlier on, reducing the amount of code that has to be upgradeable, the easier future upgrades become. The main difficulty is in performing integration and regression tests for live contracts, especially if there is some bug in the contract that must be fixed under time constraints.

finishUpgrade:

```
/**
* @dev Function to dismiss the upgrade capability
* @return True if the operation was successful.
*/
function finishUpgrade() onlyUpgradeManager public returns (bool) {
        isUpgradeFinished_ = true;
        FinishUpgrade();
        return true;
}
```

This function, executable only by the upgrade manager, disables forever the ability to perform upgrades, thus finalizing `M5Token` and `M5Logic`.

Comments

9. Ends forever the role of the upgrade manager.

View functions

View functions available for `M5Token`, `M5Logic`, `upgradeManager` and `isUpgradeFinished`, which comprise all of `MinableM5Token`’s fields.

getM5Reward:

```
/**
* @dev Calculate the reward if withdrawM5() happans on this block
* @param _miner The address of the _miner
* @return An uint256 representing the reward amount
*/
function getM5Reward(address _miner) public view returns (uint256) {
        require(M5Logic_ != address(0));
        if (miners[_miner].value == 0) {
                return 0;
        }

        // adopted from https://gist.github.com/olekon/27710c731c58fd0e0bd2503e02f4e144
        // return length
        uint16 returnSize = 256;
        // target contract
        address target = M5Logic_;
        // variable to check delegatecall result (success or failure)
        uint8 callResult;
        assembly { // solium-disable-line
                // return _dest.delegatecall(msg.data)
                // calldatacopy(t, f, s)  - copy s bytes from calldata at position f to mem at position t
                calldatacopy(0x0, 0x0, calldatasize)
                // delegatecall(g, a, in, insize, out, outsize) - call contract at address a with input
                mem[in..(in+insize))
                // providing g gas and v wei and output area mem[out..(out+outsize)) returning 0 on error
                (eg. out of gas) and 1 on success
                // keep caller and callvalue
                callResult := delegatecall(sub(gas, 10000), target, 0x0, calldatasize, 0, returnSize)
                switch callResult
                case 0
                { revert(0,0) }
                default
                { return(0, returnSize) }
        }
}
```

This view function uses a standard scheme for `delegatecall`-ing a contract, supporting reading the return data.

Comments

10. As commented in the general comments as well, it should be ensured that no state changes occur due to execution of the `delegatecall`. A method of doing this is by adding a method also using assembly, but calling `staticcall` before executing the original method that does `delegatecall`. It should also be guaranteed that the called contract does not depend on the caller and the `callvalue` (as should be the case here).

Given that the above scheme implements a “`static-delegatecall`” opcode that does not exist, and due to sophistication and associated risk of such a scheme, the inferior alternative is to manually ensure every upgrade to not change the state in methods that are intended to be defined as view (Solidity doesn’t enforce it when delegate-calling).

11. Every upgrade entails verifying that the called method returns a value within the expected size.
12. The last argument, `resultSize`, is set to 256, but it should be 32. Question is whether this figure refers to bits or bytes. As this parameter refers to an offset in the memory, which is byte-addressed, it should be specified in **bytes**. `uint256`, the expected return type, is 256 bit or alternatively 32 bytes. Therefore, the `returnSize` should be set to 32.

General comments

13. The owner of MCoin is only relevant during the distribution (crowdsale) process. Afterwards, it can only change itself, the owner, and not anything else about the contract. Still, it is a dangling field with unclear purpose following the crowdsale’s completion. Perhaps the contract should support owner’s renunciation of ownership.
    Same applies to the upgrade manager.
14. For `MinableM5Coin`, consider allowing to upgrade both `M5Token` and `M5Logic` using a single transaction, in case the updates are coupled.
15. For MCoin, consider to include logic in the wallet application that disallows or warns against multiple transactions in the same block (i.e. disallow new transactions before the previous one was mined and confirmed).
16. `M5Token` mock inherits from `BurnableToken` but doesn’t use the `burn` function. It would make sense to make `burn` non-public and call it from `swap`, but the API is different as the caller is different (`owner` in the case of `swap`).
17. The implementation where the owner of the `M5Token` contract can swap anyone’s tokens is also confusing and possibly risky. Why would the owner of the M5 token contract be allowed to perform actions on anyone’s tokens?
18. The `M5tokenMock` is also confusing because of the role of the owner. The owner is supposed to be the `M5Logic` contract, in order for the mock to work as expected (specifically `mint` and `swap`). This means these two upgradable contracts are highly coupled. Some scenarios which seem not well defined are:
    - Upgrade of `M5Logic`: Entails changing the owner of the M5 token contract after upgrading the M5 logic contract. How can it be done if the M5 logic contract has no ability to call transferOwnership?
    - A scenario where the M5 token contract is upgraded with a wrong owner, not matching the current M5 logic contract.

4. ComplianceStore contract

A `MinableM5Token` also implements a compliance store, as defined by Section 4 of the whitepaper, to allow compliance with AML-KYC policies.

The compliance store implements a simple interface for `MultihashStore`. Every update of the store emits an event. Multihash is a mechanism which allows specifying not only a hash, but also its expected length and an identifier of the hash function that generated it. This way, different policies can use it with their own hash functions.

Comments:

1. A natural question is who governs the hash function types, coming from a domain 2^8=256 in size? We expect the MCoin team to provide some initial mapping of common hash functions to codes that can be used for the compliance store in a uniform manner.
2. As there is no interaction between MCoin and the compliance store, why is the compliance store a part of the MCoin contract? Why it is not a separate contract?
3. Do M5 tokens have a separate compliance store?
4. Consider adding clearHash() function to the API to allow gas refunds.
5. An example application illustrating how `ComplianceStore` is used can help developers willing to develop applications on top of the MCoin ecosystem.

Distribution Contract

Token distribution is performed in two periods, each consisting of several time windows. The maximum number of windows is 365. Each window has a predetermined allocation of tokens that will be distributed in the time period defined by the window.

Token supply is fixed. A portion of the supply, ‘foundation reserve’, is given to the ‘foundation wallet’ upon initialization of the crowdsale.

Representation invariants:

- `Sum_w(totals[w])=Sum_u,w(commitments[u][w])`
- `totals[w] = Sum_u(commitments[u][w])`

Immutable after construction: all but `totals` and `commitments`.

Non-view functions

Constructor:

```
function MCoinDistribution (
        uint _firstPeriodWindows,
        uint _firstPeriodSupply,
        uint _secondPeriodWindows,
        uint _secondPeriodSupply,
        address _foundationWallet,
        uint _foundationReserve,
        uint _startTimestamp,
        uint _windowLength
) public
{
        require(0 < _firstPeriodWindows);
        require(0 < _firstPeriodSupply);
        require(0 < _secondPeriodWindows);
        require(0 < _secondPeriodSupply);
        require(0 < _foundationReserve);
        require(0 < _startTimestamp);
        require(0 < _windowLength);
        require(_foundationWallet != address(0));
        firstPeriodWindows = _firstPeriodWindows;
        firstPeriodSupply = _firstPeriodSupply;
        secondPeriodWindows = _secondPeriodWindows;
        secondPeriodSupply = _secondPeriodSupply;
        foundationWallet = _foundationWallet;
        foundationReserve = _foundationReserve;
        startTimestamp = _startTimestamp;
        windowLength = _windowLength;

        totalWindows = firstPeriodWindows.add(secondPeriodWindows);
        require(totalWindows <= MAX_WINDOWS);
}
```

Comments

1. Parameters to constructor should be also `uint256` as in the declaration of the contract's fields.
2. `startTimestamp` should be checked, that it does not initiate a crowdsale which already ended. This will happen if `startTimestamp <= block.timestamp – (windowLength*totalWindows)`. The more precise check is to ensure in the constructor that the first window is 0. Namely, that `windowOf(block.timestamp)` returns 0 right before the constructor ends.
3. Consider checking the identity of the foundationWallet in a 'deeper' way. Perhaps it can be checked against a registrar.

Init:

```
/**
* @dev initiate the distribution
* @param _MCoin the token to distribute
*/
function init(MinableToken _MCoin) public onlyOwner {
       require(address(MCoin) == address(0));
       require(_MCoin.owner() == address(this));
       require(_MCoin.totalSupply() == 0);

       MCoin = _MCoin;
       MCoin.mint(this, firstPeriodSupply.add(secondPeriodSupply).add(foundationReserve));
       MCoin.finishMinting();

       MCoin.transfer(foundationWallet, foundationReserve);
}
```

Accepts a properly initialized MCoin. This function is controlled by crowdsale owner. It mints the initial supply to the owner of the crowdsale, and transfers the relevant portion `foundationReserve` to `foundationWallet`. This means that the foundation wallet address owner is already controlling over the tokens.

CommitOn:

```
/**
* @dev commit funds for the given window
* @param window to commit
*/
function commitOn(uint256 window) public payable {
        // Distribution have started
        require(startTimestamp < block.timestamp);
        // Distribution didn't ended
        require(currentWindow() < totalWindows);
        // Commit only for present or future windows
        require(currentWindow() <= window);
        // Don't commit after distribution is finished
        require(window < totalWindows);
        // Minimum commitment
        require(0.01 ether <= msg.value);

        // Add commitment for user on given window
        commitment[msg.sender][window] = commitment[msg.sender][window].add(msg.value);
        // Add to window total
        totals[window] = totals[window].add(msg.value);
        // Log
        Commit(msg.sender, msg.value, window);
}
```

Performs the actual sale. The user chooses a window to commit on. It must occur after the crowdsale has already started, and the committed value must be at least 1/100th of an ether. The function updates the total commit of the sender for the window, and the total window commit. It will be possible to convert ether to MCoin tokens right after the window has closed (see withdraw function description).

Comments

4. It should be noted that the 0'th window is the first window, thus the check is (`window < totalWindows`). This may be confusing and worth documenting.
5. The function checks if the distribution started. We note that (1) the constructor could have set the distribution to start in the future; (2) the check for whether the distribution ended is set by the computation of the current window.
   A minor risk is that this may enable miners, who have some limited control over the timestamp of the block, to give priority to preferred/bribing parties. To minimize the control of the miners, which may discriminate between participants in the crowdsale, it may be preferable to have the distribution owner specifically set a boolean stating whether the distribution is active or not. Specifically, the computation of the current window may in particular be prone to miner time skewing, and may lead participants that commit early in the next window to actually commit on a window whose allocation already been depleted and just ended. The user can benefit from a view function that provides remaining time for the window, and use it to decide if it's large enough to avoid miner time-skewing issues.

Withdraw:

```
/**
* @dev Withdraw tokens after the window was closed
* @param window to withdraw
* @return the calculated number pf tokens
*/
function withdraw(uint256 window) public returns (uint256 reward) {
        // Requested window already been closed
        require(window < currentWindow());
        // The sender hasn't made a commitment for requested window
        if (commitment[msg.sender][window] == 0) {
                return 0;
        }

        // The Price for given window is allocation / total_commitment
        // uint256 price = allocationFor(window).div(totals[window]);
        // The reward is price * commitment
        // uint256 reward = price.mul(commitment[msg.sender][window]);
        // Same calculation optimized for accuracy (without the rounding of .div in price calculation):
        reward = allocationFor(window).mul(commitment[msg.sender][window]).div(totals[window]);
        // Transfer the tokens
        MCoin.transfer(msg.sender, reward);

        // Init the commitment
        commitment[msg.sender][window] = 0;
        // Log
        Withdraw(msg.sender, reward, window);
        return reward;
}
```

Converts committed ether to tokens. The conversion formula is optimized for accuracy: allocation for the current window is first multiplied by the sender's commitment in wei, then divided by the total wei committed by all users in this window.

Comments

6. If the total wei committed is very large, and the allocation for the window too small, the resulting reward may be 0 for a user whose contribution was minimal. As the allocation per window is known right after the creation of the crowdsale, there should be a cap on the total wei committed by all users, to avoid this. The reward, which is an integer number, should be guaranteed to be some minimal value.
   Another possible mitigation to this issue, is to have `withdrawAll()`, which currently iterates on all finished windows and withdraws for each, to first compute the total commit by the user over all these windows, as well as the respective total for all participating users, and multiply by the correct allocation (this depends on the number of windows withdrawn from *each* of the two periods).
7. There is a call to MCoin in this function that performs token transfer, prior to nullifying the commitments. A good rule-of-thumb is to have other contracts' code run after the state of the current contract has been fully updated. Supposedly, if MCoin allowed callback to sender in the transfer method, a malicious user could devise a contract that commits, and then withdraws

multiples of the committed ether’s value in tokens by using a recursive call to `withdraw`. While it is true that the intended MCoin contract to be deployed is not calling external contracts, the user who uses these contracts cannot know for sure which code runs in the EVM.

moveFunds:

```
/**
* @dev moves Eth to the foundation wallet.
* @return the amount to be moved.
*/
function moveFunds() public onlyOwner returns (uint256 value) {
        value = this.balance;
        require(0 < value);

        foundationWallet.transfer(value);
        MoveFunds(value);
        return value;
}
```

Handles transfer of all ether to the foundation wallet, whether these are ether paid by crowdsale participants, and whether these are ether mistakenly transferred by suiciding contracts. Thus, there are no locked funds in the distribution contract. The transfer is transparently logged.

Fallback function:

```
/**
* @dev Commit used as a fallback
*/
function () public payable {
        commit();
}
```

Performs commit on the current window.

General Comments

In the current situation, it is possible to use the same ether again to buy more tokens, by using funds in control of the foundation wallet to commit on more tokens. Ideally, owner of the distribution contract as well as the foundation wallet should be blacklisted from participating in the crowdsale. An effective way to do that, is to disallow the foundation wallet from moving funds prior to crowdsale end, and thus avoid recycling of ether for buying more tokens. The decision on whether to do this or not in the final contract is up to the team.

Non-view functions

allocationFor:

```
/**
* @dev return allocation for given window
* @param window the desired window
* @return the number of tokens to distribute in the given window
*/
function allocationFor(uint256 window) view public returns (uint256) {
       require(window < totalWindows);
       return (window < firstPeriodWindows)
                      ? firstPeriodSupply.div(firstPeriodWindows)
                      : secondPeriodSupply.div(secondPeriodWindows);
}
```

Returns the allocation for the window, depending on its period.

windowOf:

```
/**
* @dev Return the window number for given timestamp
* Each window is 23 hours long
* @param timestamp
* @return number of the current window in [0,inf)
* 0 will be returned before distribution start and during the first window.
*/
function windowOf(uint256 timestamp) view public returns (uint256) {
       return (startTimestamp < timestamp)
                      ? timestamp.sub(startTimestamp).div(windowLength)
                      : 0;
}
```

Computes the current window number based on the timestamp. The comment in the function regarding the window length is wrong; it is not 23 hours but rather parametric. This is misleading.

getAllRewards:

```
/**
* @dev returns a array filed with reward for every closed window
* a convinience function to be called for updating a GUI.
* To actually recive the rewards use withdrawAll(), which consumes less gas.
* @return the calculated number of tokens for every closed window
*/
function getAllRewards() public view returns (uint256[MAX_WINDOWS] rewards) {
       for (uint256 i = 0; i < currentWindow(); i++) {
              rewards[i] = withdraw(i);
       }
       return rewards;
}
```

A view function that computes the current withdrawn values for a user, without doing actually any change in the state (as it is a view).

Disclaimer