Harbor
Smart Contract

Mikhail Vladimirov and Dmitry Khovratovich

17th July 2018

This document describes the audit process of the Harbor Regulator smart contracts performed by ABDK Consulting.

1. Introduction

We've been asked to review the Harbor smart contract given in the commit 9d3ca.

The contracts implement a sophisticated business logic for security tokens, where owners are represented by UIDs rather than addresses, ownership is restricted in the number of shares, and transfers are restricted to whitelists. The rules are dynamic and are regulated by special kind of administrators which also rotate. The contracts REITRegulatorService and RegulatedToken are examples of implementation of these rules.

We have found that the rule system is quite complicated as it is a combination of owner-based and role-based control. It might be nontrivial to integrate it with token contracts. It is recommended to have a detailed instruction on how to use the supplementary contracts and how to assign roles, as this process is error-prone.

Regarding the regulated examples, we have found a number of issues, the major being inconsistent and incorrect counting of owners and foreign owners in certain edge cases. The details follow.

2. ServiceRegistry

In this section we describe issues related to the smart contract defined in the ServiceRegistry.sol

2.1 Suboptimal Code

This section lists suboptimal code that was found in the smart contract.

Line 49: the assignment `address oldService = service` is redundant. The event could be emitted as `LogReplaceService(service,_service)` before the assignment.

2.2 Unclear Behavior

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

Line 34: should the `_service` input value undergo the `withContract` test?

3. REITRegulatorService

In this section we describe issues related to the smart contract defined in the REITRegulatorService.sol

3.1 Major Flaws

This section lists major flaws, which was found in the smart contract.

1. Line 320-321: if `_from` is the same as `_to`, balance is non-zero and the whole balance was transferred, `subShareholder` will be false and `addShareholder` will be true. This allows anybody with tokens to increase stored numbers of shareholder, without increasing the actual number.
2. Line 327: if `_amount` and `_to balance` are 0 but `_from` balance is non-zero, the branch `addShareholder && !subShareholder` will be executed with the same effect as above.
3. Line 197: `sub` may effectively revert a transaction if the currently stored value for foreign owned shares is less than the balance of a participant, making it impossible to reset "is foreign" flag for this participant.
4. Line 514: `currForeignShares.sub(_amount)` may revert transaction if `_amount` is greater than stored number of foreignly owner shares.
5. Line 562 : the condition `fromBalance == _amount && toBalance > 0` will be true if `_from == _to`, balance is non-zero and the whole balance is about to be transferred, while such transfer will not change number of shareholders.

3.2 Moderate Flaws

This section lists moderate flaws, which were found in the smart contract.

1. Line 147, 162: the function `_setShareholderCount` may become a cause of inconsistency between the stored value for shareholders and actual number of shareholders.
2. Line 272: If `_amount==balances[][]==0`, then `shareholderCount` will still increase.

3.3 Documentation Issues

This section lists documentation issues, which were found in the smart contract.

1. Line 172: the comment is incorrect. If `address` is `bytes32` type, it is probably not an address.
2. Line 448: the comment should to be added: `new balance should be less than capped amount **or equal to**.`

3.4 Unclear Behavior

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

1. Line 111, 130: perhaps, there should be `0` instead of `1`.
2. Line 33, 40: a fractional percentage might be needed for ownership counting.. Consider using some decimals here, e.g. 18.
3. Line 225: looks like the method `onlyTokenOrAdmin(_token)` does not modify blockchain state. Perhaps, there is no need to restrict access to it.

3.5 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

1. Line 277: the function `_setShareholderCount` is executed even if shareholder count does not change.
2. Line 285: the function `_setForeignOwnedShares(msg.sender, currForeignOwnedShares)` is executed even if number of foreignly owner shared didn't change.
3. Line 331: the function `_setShareholderCount(msg.sender, currCount)` is executed even if number of shareholders didn't change.

4. RegulatedToken

In this section we describe issues related to the token defined in the RegulatedToken.sol

4.1 Moderate Flaws

This section lists moderate flaws, which were found in the token smart contract.

Line 18: using a public constant will produce getter method doing the same as standard `decimals()` method but with different name. This looks redundant.

4.2 Unclear Behaviour

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

Line 57, 129: perhaps, function `mint` and `reissue` should also call `_check`.

4.3 Suboptimal Code

This section lists suboptimal code patterns, which were found in the token smart contract.

1. Line 159: Using `msg.sender` as spender makes it impossible for regulator service to distinguish transfer attempt from `transferFrom` with `_from==msg.sender`. It would be better to pass zero spender in case of `transfer`.
2. Line 161: there should be `CHECK_SUCCESS` instead of `0`.

5. BaseRegulatorService

In this section we describe issues related to the smart contract defined in the BaseRegulatorService.sol

5.1 Critical Flaws

This section lists a critical flaw, which was found in the smart contract.

Line 270: there should be `_spender` instead of `tx.origin`. Also, `tx.origin` is easy to manipulate. One may convince somebody with spender role to send transaction to seemingly unrelated contract, that will actually perform `transferFrom` operation on some token.

5.3 Documentation Issues

This section lists documentation issues, which were found in the smart contract.

1. Line 56: it is unclear how participant address is represented as bytes32. Would be good to have a clarifying comment.
2. Line 61: event name `LogLockSet` is confusing, one may think that this event is logged only when lock is set, but not when lock is removed.

5.4 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

Line 177: the check `settings[_token].locked` should probably come first, as it is the cheapest one.

5.5 Unclear Behavior

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

- Line 165: the method `onlyTokenOrAdmin(_token)` looks like it does not modify blockchain state. Perhaps, there is no need to restrict access to it.

5.6 Arithmetic Overflow Issues

This section lists issues of token smart contract related to the arithmetic overflows.

- Line 285: the overflow is possible `**`. One may create token with more than 77 decimals to make sure all transfers are partial, but this contract will break on such token.

6. Wallet Manager

In this section we describe issues related to the token smart contract defined in the WalletManager.sol.

6.1 Unclear Behavior

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

- Line 80: the function `addUserWallet` allows identical UUIDs for different wallets. Is it really necessary?

6.2 Suboptimal Code

This section contains suboptimal code patterns, which was found in the smart contract.

1. Line 58-59: the `view` methods are usually assumed to be callable off-chain, but `onlyRole` modifier checks `msg.sender` and thus requires method to be called on-chain. Consider removing access control modifiers from `view` methods.
2. Line 109: the parameter `bytes32 _user` looks redundant. Its value could be derived from the `wallet` parameter.

7. ReissuableToken

In this section we describe issues related to the token smart contract defined in the ReissuableToken.

7.1 Unclear Behavior

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

1. Line 13: the contract already has Ownable access control mechanism. Should it also have Role-based one?
2. Line 83: perhaps, only one event is needed here.

7.2 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

1. Line 50: it might be better to use `emit LogReissuerSet` at the beginning of the method. Then the variable `oldReissuer` would not be needed.
2. Line 74: the function `reissue` always returns `true`. Maybe it should return nothing.

8. ErrorMixin

In this section we describe issues related to the token defined in the ErrorMixin.sol

8.1 Suboptimal Code

This section lists suboptimal code patterns, which were found in the token smart contract.

Line 13: the contract `ErrorMixin` looks more like `enum`. Consider using `enum` instead of it.

9. BalanceManager.sol

In this section we describe issues related to the smart contract defined in the BalanceManager.sol

9.1 Documentation Issues

This section lists documentation issue, which was found in the smart contract.

Line 16: it is unclear how `shareholder` is represented by `bytes32`. It would be better to add more details.

9.2 Suboptimal Code

This section lists suboptimal code pattern, which was found in the smart contract.

Line 51: the `view` functions are usually supposed to be callable off-chain, but `onlyRole` modifier depends on `msg.sender` and thus needs to be evaluated on-chain. Consider removing access control modifiers from `view` functions.

Line 20: three events for the balance changes look redundant. Probably, one event with old and new balances, should be enough. Also, three events are harder to track.

10. Domicile

In this section we describe issues related to the token defined in the Domicile.sol

10.1 Unclear Behavior

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

Line 10: the comment with more details is needed. Perhaps, instead of `participantAddress` there should be `address`.

10.2 Suboptimal Code

This section lists suboptimal code pattern, which was found in the smart contract.

Line 12: the field `internal` is not public and there is no getter for it. Consider adding getter.

11. AccessRestricted

11.1 Suboptimal Code

This section lists suboptimal code pattern, which was found in the smart contract.

1. Line 10: the contract `AccessRestricted` uses two access control schemas in parallel: owner-based and role-based. Consider making owner to be just one of the roles.
2. Line 17: the `string` roles are subeffective and error-prone. Consider using `enum`.

12. Our Recommendations

Based on our findings, we recommend the following:

1. Fix major and moderate issues which could damage the business logic.
2. Fix arithmetic overflow issues.
3. Check issues marked “unclear behavior” against functional requirements.
4. Refactor the code to remove suboptimal parts.
5. Fix the documentation and other (minor) issues.