HARBOR
Smart Contracts

Mikhail Vladimirov and Dmitry Khovratovich

8th March 2019

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

1. Introduction

We've been asked to review the Harbor smart contract given in a private access to the Harbor repository, tagged as `audit-3.1`.

2. RegulatedToken

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

2.1 Suboptimal Code

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

1. Line 268: the variable `serviceKey` is set via `setServiceKey` method defined in `ServiceLocatorConsumer`. This variable may be used before initialization. It would be safer to explicitly require `serviceKey` to be not empty before use, like this: `require(bytes(serviceKey).length > 0);`.
2. Line 34: both successful and failed checks are logged which is probably redundant. Consider logging only failed checks, because each successful check will lead to Transfer event anyway.
3. Line 52: keeping `_name, _symbol, RTOKEN_DECIMALS` in storage is more expensive than in bytecode. Consider using compile-time constants for name, symbol and decimals.
4. Line 81: `mint` internally performs access check on each invocation, which is redundant, because `batchMint` method is already guarded with `onlyReissuer` modifier. Also, this calls `_onMintSuccess` which in turn calls `_service()`, and `_service()` accesses another contract which is not cheap. Consider optimizing.

3. ServiceLocator

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

3.1 Suboptimal Code

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

1. Line 9: strings are expensive. Consider using hashes.
2. Line 41: the `contractAddresses[_key]` returns zero address for unknown keys, which could make hard to track errors. Consider throwing on unknown keys.

3.2 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

1. Line 13: the `key` field should be probably indexed.
2. Line 14: perhaps, the `oldAddress` is redundant. It can be retrieved from the previous event.

4. IServiceLocator

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

Line 4: the string keys are expensive. Consider using fixed-sized hashes of keys instead. And it is uncommon to declare interface methods as `external`. Consider changing to `public`.

5. IRegulatorService

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

Line 20,71: from comment unclear to say what is the value in case of plain transfer, i.e. not `transferFrom`.

6. BaseRegulatorService

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

6.1 Major Issues

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

1. Line 217: there is no need to exclude the case `_spender == _from`. One may try to bypass controls by approving to himself.
2. Line 230: if `ROLE_SPENDER` calls any malicious contract (even indirectly) this check will pass and the transfer will be enabled (even if `ROLE_SPENDER` has no clue of calling this contract). Also, this does not allow smart contracts (e.g. multisig wallets) to be valid spenders, because the smart contract may never be transaction origin.

6.2 Documentation issues

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

Line 87: the comment `Permission bits to be set` is incorrect. This method not only sets some permission bits, but rather overrides all permission bits, i.e. sets some bits and clears all the others.

6.3 Suboptimal Code

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

1. Line 41: using `uint8` instead of `uint256` just limits number of possible permissions to 8 without any performance benefits.
2. Line 128-130: the method does not modify blockchain state. Perhaps, there is no reason to limit access to it. Declaring it with `view` modifier and removing access restrictions would make it less expensive and more convenient to use.
3. Line 134: the method `_storage()` is called multiple times returning the same result. It calls another contract internally, which is not cheap. Consider calling once and caching in memory.
4. Line 230: the role `ROLE_SPENDER` is set globally, rather than per token. The permission to spend others tokens is handled differently from permission to send and receive token. Is it OK?
5. Line 248: the method `decimals()` is optional according to EIP20 standard, and the token contracts that do not support it will probably throw here. Consider handling such situation by using `call()`.
6. Line 249: the case `"Too many decimal points to compute whole token"` is not forbidden by standard. It is possible to have token with total supply less than 1 whole token and very large number decimals. Should probably count any number of such tokens as not-whole, rather than to throw.

6.4 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

Line 61,72: it would be better to rename `_enabled` to `_divisible`. The same for `_locked` in `setLocked`.

7. REITRegulatorService

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

7.1 Suboptimal Code

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

1. Line 38: the `CAP_PRECISION` should be declared as constant.
2. Line 79: the message `Got unexpected zero value` is useless because it does not contain details about what particular parameter is zero.
3. Line 127,148: the constant `uint32(100).mul(uint32(10)**CAP_PRECISION)` should be made compile-time.
4. Line 199: the method `_reitStorage()` is called multiple times here returning the same result, and internally this calls another contract, which is not cheap. Consider calling once and caching in memory.
5. Line 213,449: the value of foreign owned shares variable in storage is not guaranteed to be equal to the sum of balances of all shareholder marked as foreign. Thus, this subtraction may lead to underflow and exception. Consider setting foreign owned shares to zero in case current is less than value to be subtracted and for line 449 returning zero in case `currForeignShares` is less than `_payload`.amount.
6. Line 241: the method `onlyTokenOrController(_token)` does not modify blockchain state,perhaps there is no need to restrict access to it. Marking it with `view` modifier and removing access restrictions would make it cheaper and more convenient to use.
7. Line 253: the method `violatesShareholderMax` internally calculates new number of shareholders already calculated by `violatesShareholderMin` method. Consider calculating once and caching in memory.
8. Line 340: the interface `ERC20Basic` should be IERC20 rather than particular implementation.
9. Line 401: the operation `_computeTransferBalances(_payload)` calculates four values but only one value is used: `transferBals.newToBalance`.

10. Line 437: the `currForeignShares` may be derived from `_payload.token`, no need to pass separately.
11. Line 503: the logic of the method `shareholderCountFromBalances` is overcomplicated. Consider simplifying like this:
```
if (transferBals.newFromBalance > 0 &&
transferBals.prevToBalance == 0)
returns currCount.add(1);
else if (transferBals.newFromBalance == 0 &&
transferBals.prevToBalance > 0)
return currCoun.sub(1);
else return currCount;
```
12. Line 278-279,589: checks was already in `onMintSuccess` and `onTransferSuccess` no need to check again here.
13. Line 557: the variable `getWalletOwner(_to` is calculated twice within the same method. Consider caching in memory.
14. Line 561: the check is redundant, because `_amount` is guaranteed to be non-zero here.
15. Line 651: the `mul(100).mul(uint256(10) ** CAP_PRECISION)` should be compile-time constant.
16. Line 678-679: restricting access to `private view` method that does not modify blockchain state looks like waste of gas.

7.2 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.

1. Line 126: perhaps, there should be `uint24` instead of `uint32`. 100% with four decimals is only 1,000,000, which fits into `uint24`.
2. Line 362,379: in case new shareholders count is less than minimal allowed shareholders count, the contract denies even the transfers that actually increase (decrease for the line 379) shareholders count. Is it OK?
3. Line 424: in case new amount of foreign owned shares is greater than cap, the contract deny even transfers that actually decrease amount of foreign owned shares. Is it OK?
4. Line 529: if `currCount` is 0 because of inconsistency between `currCount` and `transferBals`, this line will throw an exception. Is it OK?

7.3 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

1. Line 512: the brackets in this line are redundant and make code looks like function call, which is confusing.
2. Line 30: the contract `REITRegulatorService` calls a number of external contracts but no documentation on these calls is present. This makes it difficult to evaluate how the code should behave.
3. Line 98,110,128: it is unclear to say what functions `setShareholderMin`, `setShareholderMax` and `setOwnershipCap` should do. Some documentation comment would be helpful.

8. HarborClaimable

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

Line 10: it is unclear to say what the contract `HarborClaimable` should do. Some documentation comment would be helpful.

9. AccessRestricted

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

9.1 Suboptimal Code

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

Line 99,120: the method `_authorized` actually combines two different methods. Consider splitting.

9.3 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

1. Line 3: the file `zeppelin-solidity/contracts/access/rbac/RBAC.sol` is not present in the most recent versions of OpenZeppelin. Is it OK?
2. Line 9: the `HarborClaimable` adds single-ownership model in addition to role-based. Is it OK?
3. Line 51: in this line constructor of a contract are called, but this contract does not directly inherit from.
4. Line 120: the name of the function `adminAuthorizeToken` is incorrect. The name `setAuthorization` is more relevant.

10. ReissuableToken

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

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

1. Line 49: the condition `Reissuer cannot be 0` is not necessary. Setting reissuer to zero address looks like a simple and transparent way to temporarily disable reissuance.
2. Line 74: this assertion may be an overkill. Other functions (e.g. `mint`) in this contract permit sending tokens to zero address.
3. Line 76-81: the modern versions of OpenZeppelin have internal method `_transfer`: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/ERC20.sol#L127
4. Line 82: two events `LogTokenReissued` and `Transfer` log the same fields.

11.RestrictedBurnableToken

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

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

1. Line 38-39: the modern versions of OpenZeppelin have internal method `_burn`: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/ERC20.sol#L127
2. Line 41: two events `Burn` and `Transfer` log the same fields.

12. AlwaysMintableToken

In this section we describe issues related to the token defined in the AlwaysMintableToken.sol.
This section lists suboptimal code patterns, which were found in the smart contract.

1. Line 36: the `mint` function always returns true, maybe it should return nothing.
2. Line 38-40: modern versions of openzeppelin have internal `_mint` method for this: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/ERC20.sol#L142.

13. BalanceManager

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

1. Line 44, 66, 81: according to RFC4122 (https://www.ietf.org/rfc/rfc4122.txt) UUID is 16 bytes long, not 32.
2. Line 52: the method `setBalance` checks for `admin` role on every invocation which is sub effective. Consider checking once before the loop. Also, this internally calls `locator.getContractAddress(serviceKey)`, i.e. other contract, and all such calls within one transaction probably return the same result. Consider calling once and caching result in memory.

14. ErrorMixin

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

Line 10: having the set of plain `uint256` constants instead of `enum` can help to prevent the situation in the warning. Currently, values in documentation comments could easily go out of sync with real values of enum constants.

15. ServiceLocatorConsumer

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

Line 12: the contract `ServiceLocatorConsumer` contains two storage variables not related to each other, and two methods, also not related to each other. Also, it contains almost no logic. The contract looks almost useless, but makes design of the whole system more complicated. Consider removing this smart contract.

16. WalletManager

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

1. Line 83: the method checks for `admin` role on every invocation which is subeffective. Consider checking only once before the loop. Also this internally calls `locator.getContractAddress` on every invocation, that probably

returns the same result being called multiple times in the same transaction. Consider calling once before the loop and caching in memory.

2. line 122-123: the method `dev` not processed the situation then some tokens will come after wallet was cleaned up by offchain controllers, but before wallet was changed.
3. Line 140: the check `_user != ZERO_UUID` looks redundant, as we also check that:
    - wallet is owned;
    - `_user` is the owner of the wallet.

17. StorageContract

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

1. Line 9: the `Claimable` is an implicit superadmin role, in addition to RBAC, which is confusing. Consider resorting to either of them.
2. Line 10: the `ROLE_CONTRACT` should be declared as constant.
3. Line 14: the method `setContractAuthorization` actually combines two methods. Consider splitting.

18. WalletManagerStorage

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

Line 32: the function return `(wallet != address(0), "Wallet address cannot be 0x")` is different from other functions. They below permit setting zero addresses.

19. REITRegulatorServiceStorage

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

19.1 Suboptimal Code

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

1. Line 34: combining the struct `CurrentState` into the same structure with `REITSettings` could make access to mappings cheaper.
2. Line 73: the modifier `external` means that when this method is called from the contract it belongs to, `msg.sender` will refer to `this` contract, rather

than to original caller. The `onlyRole` check will not probably have any sense. Consider changing to public here and for all other similar methods.

19.2 Other Issues

This section lists stylistic and other minor issues whBщich were found in the token smart contract.

1. Line 21: it might be better to describe `assuming 4 decimal places` as a fixed point value with 6 decimal places.
2. Line 65-70: the token value should be probably indexed in all events.

20. BaseRegulatorServiceStorage

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

20.1 Suboptimal Code

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

1. Line 73: the modifier `external` means that when this method is called from the contract it belongs to, msg.sender will refer to `this` contract, rather than to original caller, so `onlyRole` check will not probably have any sense. Consider changing to public here and for all other similar methods.
2. Line 74, 91, 108, 131, 146, 161, 176, 195: the method `onlyRole(ROLE_CONTRACT)` does not change blockchain state. Restricting access to it is basically waste of gas.

20.2 Other Issues

This section lists stylistic and other minor issues which were found in the token smart contract.

1. Line 52: in other similar mappings the UUID key is first and the token is second, which is error-prone.
2. Line 56-59: arguments `token` and `participant` should be probably indexed.

21. BalanceManagerStorage

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

Line 10: some documentation is needed on which permissions are given by `ROLE_CONTRACT`.

Line 49: the method `setContractAuthorization` is already implemented in base contract `StorageContract`, and this overriding method seems to be exactly the same as overridden method.

22. Our Recommendations

Based on our findings, we recommend the following:

1. Fix the major issues.
2. Check issues marked “unclear behavior” against functional requirements.
3. Refactor the code to remove suboptimal parts.
4. Fix the readability and other (minor) issues.