PUBLIC

Code Assessment

of the System contracts

Smart Contracts

November 5, 2021

Produced for

Q

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Sir or Madam,

First and foremost we would like to thank Q Blockchain for giving us the opportunity to assess the current state of their System contracts system. This document outlines the findings, limitations, and methodology of our assessment.

During the assessment and review of the fixes, most of the found issues were addressed. However, we have found new and partially fixed issues in the new version.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	3
• Code Corrected	3
Medium-Severity Findings	6
• Code Corrected	5
• Specification Changed	1
Low-Severity Findings	17
• Code Corrected	16
• Code Partially Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the System contracts repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	28 August 2021	8023c47392fdfc714f5bb492778ecb774e8b811c	Initial Version
2	8 October 2021	cc05d55cca225fa281be3b8f73a1405e7605823c	Second Version
3	14 October 2021	e00fd1422efcc3e7ed420fe76b177e7c42022cbb	Third Version
4	1 November 2021	163afd0bfe23e786176c60d9ab9203c9cf7ed141	Fourth Version

For the solidity smart contracts, the compiler version `0.7.6` was chosen. Following files from repository contracts folder were part of the assessment scope:

```
/common/AddressStorage.sol
/common/AddressStorageStakes.sol
/common/AddressStorageStakesSorted.sol
/common/CompoundRateKeeper.sol
/common/CompoundRateKeeperFactory.sol
/common/AddressStorageFactory.sol
/tokeneconomics/PushPayments.sol
/ContractRegistry.sol
/defi/BorrowingCore.sol
/defi/DefiParams.sol
/defi/ForeignChainTokenBridgeAdminProxy.sol
/defi/GSNPaymaster.sol
/defi/LiquidationAuction.sol
/defi/oracles/FxPriceFeed.sol
/defi/Saving.sol
/defi/SystemBalance.sol
/defi/SystemDebtAuction.sol
/defi/SystemSurplusAuction.sol
/defi/token/StableCoin.sol
/defi/TokenBridgeAdminProxy.sol
/governance/AParameters.sol
/governance/ASlashingEscrow.sol
/governance/constitution/ConstitutionVoting.sol
/governance/EmergencyUpdateVoting.sol
/governance/experts/AExpertsMembership.sol
/governance/experts/AExpertsMembershipVoting.sol
/governance/experts/AExpertsParametersVoting.sol
/governance/GeneralUpdateVoting.sol
/governance/rootNodes/RootNodesSlashingVoting.sol
/governance/rootNodes/Roots.sol
/governance/rootNodes/RootsVoting.sol
```

```
/governance/validators/Validators.sol
/governance/validators/ValidatorsSlashingVoting.sol
/governance/VotingWeightProxy.sol
/tokeneconomics/DefaultAllocationProxy.sol
/tokeneconomics/QHolderRewardPool.sol
/tokeneconomics/QHolderRewardProxy.sol
/tokeneconomics/QVault.sol
/tokeneconomics/RootNodeRewardProxy.sol
/tokeneconomics/SystemReserve.sol
/tokeneconomics/ValidationRewardPools.sol
/tokeneconomics/ValidationRewardProxy.sol
```

2.1.1 Excluded from scope

Any contracts not mentioned above. Mock and testing contract that might rely on scope contracts. Imported libraries are assumed to behave according to their specification and are not part of the assessment scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Q Blockchain is a Ethereum based chain with a delegated proof of stake (DPoS) consensus mechanism, on-chain governance framework, built-in stablecoin system, and numerous other features. Majority of those system elements are implemented as on-chain smart contracts, that interact with each other. Native token of Q Blockchain is called `Q` token.

2.2.1 Governance

In System contracts, the DPoS is enforced by two types of entities:

- Validator nodes: they are responsible for the blocks validation. Holders of the `Q` token can delegate a stake to those nodes. Anyone in `Q`'s ecosystem can enter the validators list, provided they stake a sufficient amount. Validators are tracked in Validators.sol contract using 2 lists: the long list and the short list. The long list contains all addresses that placed some stake in the `Validators` contract. The stake increases their voting weight inside the `VotingWeightProxy` contract. The short list contains the top N validators ranked by stake. They are split into 3 groups: active validators, standby liquidators and backup validators. Only standby and active validators are rewarded for participation in consensus. It is assumed that a majority of the validators behaves correctly, and misbehaving validators get their stake slashed by the root nodes.
- Root nodes: they are responsible for the compliance of the validator nodes. Root nodes can also propose to slash another root node if it misbehaves. Any `Q` token holder can propose to add or remove a root node and every user of the system can take part in the vote. The membership of roots is tracked via the `Roots` contract. Any address can place a stake in the `Roots` contract, and that stake increases the voting weight inside the `VotingWeightProxy` contract. Roots can be added, replaced or removed via a vote on a `RootsVoting` contract. Any user of the system can participate in such votes with their weight inside the `VotingWeightProxy`. Roots can slash committed stake of Validators in a 2 stage weighted vote. First, the slashing proposal needs to be created and approved at the `ValidatorSlashingVoting` contract. Second, the the slashed victim can object the decision proposal, in which case a second vote needs to pass with a simple majority. The *RootNodesSlashingVoting* contract operates similarly. Proposals there can be created

only by roots, but can be voted by any network participant that has weight inside the `VotingWeightProxy`. Votes on both the `ValidatorSlashingVoting` and `RootNodesSlashingVoting` can be vetoed by a simple majority of the root nodes. It is assumed, that a majority of the roots is well behaving and actively participating in the votes.

The `Q` ecosystem is ruled by a so called `Constitution`, a set of predetermined rules and parameters such as the maximum number of root nodes, or time periods for votes for example. Anybody can propose changes to the constitution via the *ConstitutionVoting* contract, and any `VotingWeightProxy` weight holder can vote with his weight.

Network participants can also propose and vote on the `GeneralUpdateVoting` and `EmergencyUpdateVoting` contracts. While tracking of the proposal's status is done on-chain, the successful proposal has no immediate effect on network and users **have to trust** that eventually the proposal will be executed correctly.

2.2.2 Expert parameters

- EPQFI : Expert Panel for Q Fees and Incentives
- EPDR : Expert Panel for DeFi and Risk

Parameters of the native token economics and DeFi platform are stored in two contracts, `EPDR_Parameters` and `EPQFI_Parameters`. Those parameters can be respectively adjusted by experts via a vote on the EPDR_ParametersVoting and EPQFI_ParametersVoting contracts. Acceptance of the proposal needs a quorum greater than the constitution defined limit and a majority of the votes.

2.2.3 Contract Registry

The contract registry is the central contract of the system. It contains the addresses for the different core elements of the ecosystem, like `SystemBalance`, `StableCoin` or `VotingWeightproxy`. The maintainer of the registry, who is assumed to be a multisig wallet, is able to update those addresses. The maintainer has a privileged role and is trusted to behave correctly.

2.2.4 DeFi application

The *Q* blockchain has a built-in DeFi application that allows users to lock eligible assets and mint synthetic STC assets in return. The STC assets always need to be backed by a collateral whose value should exceed the value of minted tokens. If the value of collateral falls below a threshold, the on-chain liquidation auction can start. The estimation of collateral value is done using price oracles. The liquidation auction allows network participants to make bids and win the underlying collateral.

Users can lock synthetic STC assets in a `Saving` contract, which earns interest coming from borrowing and liquidation fees.

With time, the system can accumulate surplus or debt due to liquidation bids or savings payouts. Those values are tracked by the `SystemBalance` contract. The surplus or debt can be auctioned off and system balance can be restored. For surplus auction, the revenue from auctions is sent to the `DefaultAllocationProxy` contract, which later distributes the surplus between a list of beneficiary addresses. For debt auctions, the `Q` system can sell its reserves to close the outstanding debt.

For all auctions, each subsequent bid is required to be higher than the previous bid.

2.2.5 QVault

`Q` blockchain users can deposit their `Q` tokens into a ERC677 `QVault` token. The `QVault` enables users to profit from the `Q` reward pool interest rate, which is set by a EPQFI panel. The payouts for this interest rate come from the `QHolderRewardPool`. In addition, the `QVault` allows users to delegate stake (balance of `QVault`) to multiple validators. These delegations are recorded via `ValidationRewardPools` contract, that keeps track of compound rate for each validator and increases it when rewards are paid out. The `QVault` also supports time locked deposits, which allow user to withdraw or transfer more tokens linearly with time.

2.2.6 Rewards distribution

Funds that end up on the `DefaultAllocationProxy` (e.g., from block rewards, `SystemSurplusAuction` or `SlashingVoting`) are distributed between numerous beneficiaries with corresponding share for each beneficiary. `RootNodeRewardProxy`, `ValidationRewardProxy` and `QHolderRewardProxy` are assumed to be among beneficiaries.

`RootNodeRewardProxy` distributes the reward equally between all roots.

`ValidationRewardProxy` distributes the rewards between active and standby validators. A share of each validator's reward goes to the `ValidationRewardPools`. This way QVault stakers later profit from their delegations.

`QHolderRewardProxy` distributes part of the reward to the `SystemReserve` contract and the rest to the `QHolderRewardPool`. The proportion between those params is set by the EPQFI panel via `Q_reserveShare parameter`.

2.2.7 Q-less transactions

With help of GSN subsystem, the `Q` blockchain allows users to execute transactions on chain without holding the native token for gas payments. This is done with help of GSNPaymaster contract, that allows to execute transactions for other users, if they are meant for certain whitelisted targets.

The block producers on the Q blockchain are assumed to be not malicious, meaning that they don't skip, reorder or sandwich other transactions. It is also assumed that system parameter updates (e.g. rates) are regularly triggered.

Furthermore, in the findings section we have added a version icon to each of the findings to increase the readability of the report.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. We draw attention to the fact that due to inherent limitations in any software development process and software product an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third party infrastructure which adds further inherent risks as we rely on the correct execution of the included third party technology stack itself. Report readers should also take into account the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severities. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation
- Trust: Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Compound Rate normalizeAmount Code Partially Corrected

5.1 Compound Rate `normalizeAmount`

Design Low Version 1 Code Partially Corrected

The `normalizeAmount` returns values that are greater or equal to `target * decimals / rate`, but still denormalizes to the same target value. For example, target 3, rate 123 and decimals 1000 yields 25. The true value, if computed in the domain of real numbers, would be 24.39. Thus, it can be expressed, that the `normalizeAmount` performs rounding up of the result for some cases. Also, `normalizeAmount(300) == 2440` which is smaller than 100 deposits of `normalizeAmount(3) == 25`. This has few effects on the systems, that depend on CoumpoundRateKeeper.

- Saving `_checkBalance` function calls can fail and system will be rendered unusable. This can happen during the normal operation of the system. Saving contract users won't be able to deposit and withdraw funds from the contract. The rounding can effectively cause Denial of Service failure on this contract.
- QVault `_checkBalanceInvariant` can fail for the same reason and render contract DoSed.
- Saving contract can have not enough tokens to cover the deposits. Since all deposits are stored as normalized values and rounded up, all users get more tokens than they can claim from the system.
- Saving contract rewards users with smaller deposits
- BorrowingCore penalized the users, by rounding up their debt.
- The BorrowingCore will collect more fees but users also will get less liquidation coins if fee is high enough to cover both.

In addition, the update is done via a loop, that is not executed more than once, while according to fuzzing tests it never runs more than once for the domain of numbers that contract should work with.

Code partially corrected:

The ineffective loop was removed in favor of simple if condition. On repetitive deposit to the same address on the QVault the user balance loses some small values due to division with truncation in denormalization function. This loss should in most cases compensate the gain from the rounding up. Overall all mentioned problems can be mitigated by some extra funds deposits. BorrowingCore behavior also while penalizing certain parties by tiny amounts, rewards the system health.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	3

- Inconsistent extendLocking Allows Multiple Votes Code Corrected
- Multiple Votes by Delegation Code Corrected
- Owner Not Initialized Code Corrected

Medium-Severity Findings	6

- Expired Slashing Proposals Are Not Purged Code Corrected
- FxPriceFeed setExchangeRate Timestamp Code Corrected
- Gas Heavy Operation on Foreign Chain Specification Changed
- Inconsistent Liquidation Full Debt Due to Payback Code Corrected
- Price Decimals and Token Decimals May Differ Code Corrected
- ValidationRewardPools _updateCompoundRate Code Corrected

Low-Severity Findings	16

- ASlashingEscrow Decision Reordering Code Corrected
- Compiler Version Not Fixed and Outdated Code Corrected
- ContractRegistry Erasing Key Code Corrected
- Corruptible AddressStorageStakesSorted Code Corrected
- FxPriceFeed Can Have No Maintainers Code Corrected
- GSN Version String Code Corrected
- Inconsistent Liquidation Full Debt Due to Outdated Debt Code Corrected
- Inconsistent System Debt Auction Start Condition Code Corrected
- Inefficient Code Code Corrected
- Long pendingSlashingProposals Attack Code Corrected
- QVault updateCompoundRate Precision Loss Code Corrected
- Solc Pragma Code Corrected
- Specification Mismatch Code Corrected
- SystemSurplusAuction Bid Reentrancy Code Corrected
- ValidationRewardProxy Allocate Potential Overflow Code Corrected
- Validators Can Alter Delegator Share Code Corrected

6.1 Inconsistent extendLocking Allows Multiple Votes

Correctness High Version 2 Code Corrected

`VotingWeightProxy.extendLocking` only assigns `_lockNeededUntil` to `lockedUntil[_who]`. To be consistent, it should be the max between the new and the old value. Otherwise the user can manipulate the unlock times by voting on a proposal with smaller end time. In addition, the manipulation with unlock times enables user to transfer out the funds earlier and perform the attacks similar to on in Multiple Votes by Delegation, where users tokens can be reused to contribute to the same vote multiple times.

Code corrected:

`VotingWeightProxy.extendLocking` now yields the max value between `_lockNeededUntil` and `lockedUntil[_who]`.

6.2 Multiple Votes by Delegation

Design High Version 1 Code Corrected

Upon `VotingWeightproxy.announceUnlock` and `VotingWeightproxy.unlock`, no check is done to verify that the voting agent is not currently locking the delegated amount. This enables an attack where it is possible to vote multiple times with the same `Q`.

Here is the attack scenario: A_i and V_i are accounts controlled by attacker.

1. A_i : QVault.lock(X)
2. A_i : VWP.announceNewVotingAgent(V1) and VWP.setNewVotingAgent, can do it in one go because `getLockeduntil` will return 0 since A_i did not vote
3. V_i : vote on proposal, gets a lock on its own lockInfo
4. A_i : QVault.announceUnlock(X) and QVault.unlock(X), can do it in one go since A_i did not vote (no lock)
5. A_i : Qvault.transfer(A_{i+1}, X)
6. goto 1. with i = i+1

Code corrected :

Lock time is now tracked only once per user, previously it was once per locking contract and per user. Now both `announceUnlock` and `unlock` now take into account the max time between user's own time lock and its voting agent's time lock, this mitigates the attack described above.

6.3 Owner Not Initialized

Design High Version 1 Code Corrected

Some contracts inherit Ownable and Initializable, but do not assign the owner in the `initialize` function. If such contracts are deployed as a proxy, the owner field will be uninitialized (stays address 0) and owner functionality would be unusable.

- GSNPaymaster inherits Initializable and BasePaymaster. BasePaymaster inherits Ownable. Function `initialize` of the GSNPaymaster only assigns value to `stc` field. The `relayHub` field can only be set by owner.
- ForeignChainTokenBridgeAdminProxy does not set owner in `initialize`.
- TokenBridgeAdminProxy is Initializable and Ownable. Owner is not initialized. Also, the Ownable functionality is not used anywhere.
- ExpertsMembership does not initialize owner. Also, the Ownable functionality is not used anywhere.

Code corrected:

Q Blockchain has done following fixes for the issues:

- Function `initialize` was removed from GSNPaymaster. The logic from it was moved to the constructor.
- Now the contract extends OwnableUpgradeable contract of openzeppelin library. The `initialize` functions calls `_Ownable_init`, that sets the owner.
- Now the contract extends OwnableUpgradeable contract of openzeppelin library. The `initialize` functions calls `_Ownable_init`, that sets the owner.
- Now the contract extends OwnableUpgradeable contract of openzeppelin library. The `initialize` functions calls `_Ownable_init`, that sets the owner.

6.4 Expired Slashing Proposals Are Not Purged

Design Medium Version 1 Code Corrected

When `purgePendingSlashings` is called (`Validators` and `Roots`), only proposals with state `REJECTED` or `EXECUTED` are deleted. The proposals with state `EXPIRED` are kept in the `pendingSlashingProposals` list and their slashing amount is always kept into account when calculating the pending slashing amount. Thus, preventing the `Roots` and `Validators` to withdraw this amount that they should be able to withdraw, locking it forever.

Code corrected :

RootNodesSlashingVoting and ValidatorsSlashingVoting now define `slashingAffectsWithdrawal` functions. The `slashingAffectsWithdrawal` function includes a check that slashing proposal is not in `EXPIRED` state.

6.5 FxPriceFeed setExchangeRate Timestamp

Design Medium Version 1 Code Corrected

The exchange rate on FxPriceFeed is set by `setExchangeRate` function and recorded timestamp is taken from the block. Since the transactions can be delayed and reordered or put to the chain earlier than needed, the rate can be outdated by the time the block is mined. The recorded timestamp can give unreliable information about the rate status. Some approaches, like Maker price oracles, ensure that new price values propagated from the Oracles are not taken up by the system until a specified delay has passed.

Code corrected:

Field `pricingTime` was added to the FxPriceFeed contract.

6.6 Gas Heavy Operation on Foreign Chain

Design Medium Version 1 Specification Changed

`ForeignChainTokenBridgeAdminProxy` is supposed to be deployed on the Ethereum mainnet, thus its gas consumption is critical. The current complexity of the `updateTokenbridgeValidators` is actually `O(m*n)`, where `m` is length of old list and `n` is length of new list. Complexity can be reduced to `O(m+n)` if all old values in list were replaced by new list values. Also, the number of calls to other contracts should be minimized. Currently, a lot of calls to `bridgeValidators` contract are done.

Specification corrected:

Q Blockchain wants to use `IBridgeValidators` interface implementation as it is without any modifications, since it allows them easier integration with existing tokenbridge code. With this requirement, current solution is sufficient. In addition, the `O(m*n)` complexity loop is done to lower the number of calls between `ForeignChainTokenBridgeAdminProxy` and `IBridgeValidators` contracts. According to Q Blockchain tests, this lowers the overall gas consumption.

6.7 Inconsistent Liquidation Full Debt Due to Payback

Correctness Medium Version 1 Code Corrected

Owner of the Vault that is being liquidated can call `payBackStc` and repay STC after the liquidation process has started. This can lead to potential problematic scenario:

Collateralization ratio = 150% Liquidation ratio = 125% Liquidation fee = 2%

1. Vault owner has for 150 worth of collateral and 100 worth of STC.
2. Collateral value drops to 120, liquidation is opened with `liquidationFullDebt = 100`. At that point the fee should be 2. Highest bid is 105.
3. Vault owner pays back some its debt over liquidation ratio, so now the vault has for 120 worth of collateral and 88 worth of STC. Vault owner cannot call `liquidate` because the liquidation ratio does not allow this.
4. Liquidation is executed, `liquidationFullDebt` is still 100 but should be 88 by now. After liquidation, liquidator got his 120 worth of collateral token, system had its fee of 2 and user only got 105 - (100 + 2) = 3. So, in the end vault owner has lost more STC than what he should have with the liquidation. System would also burn more STC token than needed, compensating the `liquidationFullDebt` that should have changed due to payback.

Code corrected :

A modifier has been added to prohibit debt payback when vault is being liquidated.

6.8 Price Decimals and Token Decimals May Differ

Design Medium Version 1 Code Corrected

When computing the collateralization ratio in `BorrowingCore` the formula uses the price from the oracle along with the decimals of the collateral. It is an issue because there is no guarantee that the oracle price will have the same decimals as the associated collateral. The `decimalPlaces` of the FxPriceFeed should be used instead.

Code corrected:

Instead of using getDecimals, Q Blockchain uses `decimalPlaces` in BorrowingCore in functions _getColRatio and getVaultStats.

6.9 ValidationRewardPools `_updateCompoundRate`

Design Medium Version 1 Code Corrected

In attempt to improve the precision, the `_updateCompoundRate` tries to update the compound rate of the validator rate keeper with `balance - reservedForClaim - 1` tokens. Then, `denormalize(newRate, stake) - denormalize(oldRate, stake) + 1` tokens are be used to increase the `reservedForClaim` variable.

Assume following starting point of the system: oldRate == 1 stake == 200 balance == 400 reservedForClaim == 0

The `_updateCompoundRate` will update the compound rate keeper with `400-0-1 == 399` amount. Due to integer division truncation of the result, the newRate will be 2. New value for `reservedForClaim` will be `400-200+1==201`. But the stakers would be able to get only 200 tokens out from this update. This 1 token difference will not be claimable by anyone and such discrepancies will be slowly accumulating the ValidationRewardPools system.

If compound rate was updated with `balance - reservedForClaim == 400` tokens, `newRate` would be `3` and `denormalize(newRate, stake) - denormalize(oldRate, stake)==400` could have been distributed. `reservedForClaim` would have become 400 too.

If balance where == 200, the `balance - reservedForClaim - 1` would be 199, that is not enough to increase the rate. Meanwhile the solution without the `-1` would have increased the rate by 1.

To conclude, the `balance - reservedForClaim - 1` `_updateCompoundRate` algorithm slowly accumulates the errors and delays in some cases the distribution of the tokens. With time the accumulated errors can drive the denormalized stake and `reservedForClaim` values more apart and can prevent the payout of the rewards, since the `reservedForClaim` values will be greater than they should have been, if computation were done in the domain of real numbers.

Code corrected:

The sub(1) add(1) approach was dropped. Payable method `reserveAdditionalFunds(address _validator)` that increases the validators balance and reservedForClaims fields by the transferred value was added. It allows to compensate for rounding up errors if they occur. Since the accumulation error speed is not higher then number of delegators * number of updates 1 extra Q token with 18 decimals should compensate error for quite awhile.

6.10 ASlashingEscrow Decision Reordering

Design | Low | Version 1 | Code Corrected

The `proposeDecision` and `recallProposedDecision` can be called numerous times on ASlashingEscrow. Roots that want to `confirmDecision` cannot be sure what decision they are confirming due to the race condition between `confirmDecision` and `recallProposedDecision/proposeDecision`. Since the order of those transactions can vary, pending decision might be changed by the time confirmation arrives.

Code corrected:

The `confirmDecision` function takes extra decision hash argument, that solves the problem with reordering.

6.11 Compiler Version Not Fixed and Outdated

Design | Low | Version 1 | Code Corrected

The solidity compiler is not fixed in the code. The version, however, is defined in the `truffle-config.js` to be `0.7.6`.

In the code the following pragma directives are used:

```
pragma solidity ^0.7.0;
```

Known bugs in version `0.7.6` are:

https://github.com/ethereum/solidity/blob/develop/docs/bugs_by_version.json#L1509

More information about these bugs can be found here:

https://docs.soliditylang.org/en/latest/bugs.html

At the time of writing the most recent Solidity release is version `0.8.7` which contains some bugfixes.

Code correct :

Solidity compiler version has been fixed to `0.7.6` in every file, this this the last version before breaking changes of version `0.8.0`.

6.12 ContractRegistry Erasing Key

Design | Low | Version 1 | Code Corrected

Any given string key for address cannot be purged from ContractRegistry. Function `contains` always checks that address != 0. But `_setAddress` function does not allow address to be 0. Thus, unused key address will always be contained by this contract. There is no dedicated function for purging the addresses. In addition, keys are pushed to the storage `keys` array, but never can be deleted.

Code corrected:

Functions `removeKey` and `removeKeys` were added.

6.13 Corruptible AddressStorageStakesSorted

Design | Low | Version 1 | Code Corrected

AddressStorageStakesSorted contract uses linked list to manage the sorted stakes of addresses. Linked list implementation relies on `HEAD==address(0)` and `TAIL==address(1)` constants It is possible to inject `HEAD` or `TAIL` wherever in the linked list, allowing the owner to manipulate the order of the elements. The issue severity is limited, because only the `Validators.sol` contract is using the AddressStorageStakesSorted and only message senders can add themselves to this list. However, if the contract is used in another way than the `Validators.sol` contract does, the corruption of the sorted linked list could lead to severe issues.

Code correct :

A check that prohibits `HEAD` or `TAIL` to be added in the list has been added.

6.14 FxPriceFeed Can Have No Maintainers

Design | Low | Version 1 | Code Corrected

Function `leaveMaintainers` in FxPriceFeed contract does not check that there are left maintainers after the execution of this function. ContractRegistry performs such check in the same function.

Code corrected:

Check was added similar to ContractRegistry, that prevents the last maintainer from leaving.

6.15 GSN Version String

Correctness | Low | Version 1 | Code Corrected

Versions of contracts that enable the GSN functionality should reference the GSN version used 2.2.2. Currently the returned version strings are not correct. While there are no consequences on smart contract level (versions are not checked), the font end libs can have problems with compatibility.

- `StableCoin.versionRecipient` returns "0.0.1"
- `GSNPaymaster.versionPaymaster` returns "0.0.1"

Code corrected :

The version returned by `StableCoin.versionRecipient` and `GSNPaymaster.versionPaymaster` is `2.2.0` now.

6.16 Inconsistent Liquidation Full Debt Due to Outdated Debt

Correctness Low Version 1 Code Corrected

BorrowingCore vault `liquidationFullDebt` value is updated every time the `liquidated` function is called. This function can be called by anyone and LiquidationAuction calls is once during the `startAuction`. Nothing prevents the `liquidated`, as long as the vault is still undercollateralized. But there is no incentive to do so for anyone. In addition, the `liquidationFullDebt` saved in the beginning of the liquidation will be smaller than the up-to-date value of debt. The collateral interest rate grows constantly and actual debt at the end of Auction execution will be higher than `liquidationFullDebt`. The difference depends on duration of liquidation auction and interest rate on collateral. All values that depend on `liquidationFullDebt` will be affected by this discrepancy. For example, the liquidation fee that is defined as a percent of `liquidationFullDebt` will be smaller than needed and thus, the generated surplus of the system will be smaller than defined %.

Code corrected:

The `liquidate` function cannot be called when liquidated. Thus, the `liquidationFullDebt` cannot be updated, once it is set. The collateral interest rate growth won't affect the debt and according to Q Blockchain, it is intended behavior.

6.17 Inconsistent System Debt Auction Start Condition

Correctness Low Version 1 Code Corrected

From `SystemBalance.getBalanceDetails`, auction can begin if

```
systemBalance.getDebt() >= _params.getUint(stc.debtThreshold())
```

In `SystemDebtAuction.startAuction`, auction can begin if

```
_systemBalance.getDebt() > _params.getUint(stc.debtThreshold())
```

Note >= vs > difference.

Code corrected :

Both conditions are now strict inequality >.

6.18 Inefficient Code

Design Low Version 1 Code Corrected

Some code has no effect, is redundant, or simply inefficient. Removing or changing it can increase code readability and save some gas as well.

Examples

- `AddressStorage.mustRemove/mustAdd` :`AddressStorage.remove/add` already has the `onlyOwner` modifier
- `AddressStorage.size()` : `array.length` has already type `uint256`, further casting to `uint256` has no effect
- `AddressStorageStakesStore.updateStake()`: `addrStake[_addr] = _stake;` should be moved after the `if(_stake==0){}` block, otherwise on a 0 stake contract will write 0 in the slot, then delete the entry.
- `EmergencyUpdateVoting._vote()` : the first two `require` statements check the same property and have different error messages
- `QVault.withdrawTo()` : the check for user balance is already done in `_subFromBalance`
- `QVault._subFromBalance()` : upon `_targetBalance` computation, the `SafeMath` library not needed, the check for `_amount <= balanceOf(_owner)` has already been done
- `QVault._subFromBalance()` : `balanceOf` is called 3 times, while the result can be queried only once and later stored in memory variable.
- `ValidationRewardProxy.PayInformation` : this struct contains `bool ok`, which is assigned once but never used or returned
- `ASlashingEscrow.Decision`, `SystemSurplusAuction.AuctionInfo`, `SystemBalance.SystemBalanceDetails` : those structs can be optimized for tight packing
- `Validators` if `currentWithdrawal.amount` is 0, the entry validatorsInfos[msg.sender].withdrawal is first deleted and then reinitialized again.
- `RootsVoting._equals()` : function is never used
- `SystemSurplus.bid()` : `_auction` is already in storage, rewriting it to `auctions[_auctionId]` is not needed and gas heavy
- `SystemBalance.getBalanceDetails()` : a condition evaluation can be saved on average by writing this block in a `if - else if - else` style, with the most validated condition first
- `SystemDebtAuction.execute()` : call to `_checkBalances` has no effect since auction status is now `CLOSED`
- `TokenBridgeAdminProxy` : is `Ownable` but the functionality is never used
- `VotingWeightProxy.extendLocking()` : the loop may update `lockInfo.lockedUntil` for each tokenLockSource every time it is called. So, every source will have the same value for their `lockInfo.lockedUntil`, a unique lockedUntil per user would be more gas efficient
- `FxPriceFeed` can have fields defined as immutable.
- `CompoundRateKeeperFactory` and `AddressStorageFactory` can deploy minimal proxies for implementations and not the complete contract.

Code corrected:

- The redundant check is removed from `AddressStorage.mustRemove/mustAdd`
- Redundant casting removed
- The `_stake == 0` is handled properly now.
- The redundant check was removed.
- The redundant check was removed.
- The redundant calls were removed.
- The `ok` field was removed from struct.

- The field definitions were rearranged, to profit from tight packing.
- The deletion was removed.
- `RootsVoting._equals()` is removed.
- The redundant rewrite was removed.
- The conditions were rewritten in more optimal way.
- `_checkBalances` was moved to the beginning of `SystemDebtAuction.execute()` function.
- `_checkBalances` was moved to the beginning of `SystemDebtAuction.execute()` function.
- `fallback` function is onlyOwner now.
- The loop was removed.
- State variables that could be immutable are now immutable
- Proxies are used for the implementations.

6.19 Long pendingSlashingProposals Attack

Design | Low | Version 1 | Code Corrected

Malicious root can create numerous proposals on ValidatorsSlashingVoting or RootNodesSlashingVoting and make `pendingSlashingProposals` entries on Validators or Roots too long for gas limit to be executed. This will brake withdrawal functionality on those contracts for a given Root/Validator. Since the root is trusted role, the chance of such attack is considered low.

Code corrected :

Check for already pending slashing proposal pair (victim, proposer) has been added in `RootNodesSlashingVoting` and `ValidatorsSlashingVoting` upon `createProposal`.

6.20 QVault `updateCompoundRate` Precision Loss

Design | Low | Version 1 | Code Corrected

Following computations are performed in the `updateCompoundRate` function of the QVault:

```
uint256 _accruedReward = aggregatedNormalizedBalance.mul(_newRate.sub(_oldRate)).div(getDecimal());
IQHolderRewardPool(registry.mustGetAddress("tokeneconomics.qHolderRewardPool")).requestRewardTransfer(
    _accruedReward
);
```

The resulted `_accruedReward` variable will have more error, then the difference of two denormalized values, calculated with different rates. The requested reward will be smaller due to the integer division truncation error. With time, the error can accumulate and break invariant of the contract. In addition, `_checkBalanceInvariant` is not performed after the `updateCompoundRate`.

Code corrected :

The Q Blockchain provided following fixes:

- call `_checkBalanceInvariant` at the end of updateCompoundRate

- change calculation of `_accruedReward` to our standard pattern:
    - denormalize aggregated balance with old rate
    - update compound rate
    - denormalize aggregated balance with new rate
    - `_accruedReward` must be the diff between the two

6.21 Solc Pragma

Design | Low | Version 1 | Code Corrected

Current contract pragma directive permits quite many versions of the compiler: `pragma solidity ^0.7.0;` Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs.

Code corrected:

Solidity pragma is set to fixed `0.7.6` for all Smart Contracts. This is the latest version of Solidity 0.7 major version compiler.

6.22 Specification Mismatch

Correctness | Low | Version 1 | Code Corrected

Code does not match with the specification.

Examples:

1. `AddressStorageStakes.add()` : spec says `@param _addr address whose stake will be decreased` and `@param _stake amount of decreasing` but it should be increase instead of decrease
2. `AddressStorageStakes.sub()` : `data.stake >= _stake` check does not match with the error message
3. `FxPriceFeed` : `@title Root nodes voting` is wrong
4. `BorrowingCore.withdrawCol()` : `userVaults[msg.sender][_vaultId].colAsset > _amount` check does not match with the error message
5. `QHolderRewardPool.requestRewardTransfer()` : return specs to not match with code, there is no `_unsatisfyableClaims` and if the `amount` is too big, call just reverts and does not return `0`.
6. `SystemBalance.increaseDebt()` : PDF documentation says only liquidation auction and saving should be allowed to increase debt. Eligible contracts are not fixed, could be more contracts than those two
7. `ConstitutionVoting.shouldExist()` : spec says `@dev Internally counts the vetos percentage` but modifier only checks for proposal existence
8. `VotingWeightProxy.announceUnlock()` : spec says function should throw error 028002 if _amount = 0 but no check is done

9. `VotingWeightProxy.unlock()` : spec says we can only unlock previously announced amount via announceUnlock, but it is possible to unlock more than announced

---

Code corrected :

1. specs updated
2. error message updated
3. specs updated
4. error message updated
5. specs updated
6. new modifier has been added to check for `LiquidationAuction` and `Saving`
7. specs updated
8. specs updated
9. specs updated

6.23 SystemSurplusAuction Bid Reentrancy

Security Low Version 1 Code Corrected

Function `bid` in SystemSurplusAuction calls `PushPaymaster.safeTransferTo` to bidder address. After the call some `_auction` storage variable is reassigned. While PushPaymaster call has a limited 30000 gas, this can be still enough for reentrancy. Reordering of call and storage assignments can close this potential vulnerability.

Code corrected:

The order of operations was changed. No storage reads/writes happen after the `PushPaymaster.safeTransferTo` call.

6.24 ValidationRewardProxy Allocate Potential Overflow

Design Low Version 1 Code Corrected

The `allocate` function has following computations:

```
p.delegatorReward = p.balance.mul(shortList[i].amount).mul(p.qsv).div(p.totalStake).div(decimal);
p.validatorReward = p.balance.mul(shortList[i].amount).mul(decimal.sub(p.qsv)).div(p.totalStake).div(decimal);
```

The max value for uint256 is close to 10^77. Balance has 18 decimals, amount 18 decimals as well. The qsv or decimal-qsv is a value of 10^27 magnitude. Overall it makes 10^63 just for decimals calculations. Keeping in mind the potential big values for shortList amounts and distributed p.balance, the overflow can occur and block all Validator reward payouts from execution.

Code corrected :

Computation that are prone to intermediate overflows are now done using the function `mulDiv` of library FullMath from Uniswap V3.

6.25 Validators Can Alter Delegator Share

Trust | Low | Version 1 | Code Corrected

The `setDelegatorsShare` function of ValidationRewardPools allows validators to set a percent, that will go to delegators from the rewards that validator can get. This action can be done without any time constraints and prior announcement. Validator can even sandwich the call to rewards distribution function between two `setDelegatorsShare` calls that will result in harder to notice of lowering the profits of delegators.

Code corrected :

Q Blockchain added event `DelegatorsShareChanged` that allows users to easily identify misbehaving validators. The own stake of validator is much higher than the potential profit from reward distribution and since the own stake can be slashed for misbehavior, the issue is considered as resolved.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Constructor and Initializer

Note Version 1

Some contracts, e.g. AddressStorage defines both constructor and function `initialize`. Quite often developers duplicate the logics of `initialize` inside constructor and define constructor itself with initializer modifier. In your case you don't do it, but the creation and initialization of such contracts is consistent and done in a safe way. We wanted to let you know that the current pattern can easily lead to issues, if contract will be used directly without proxy and without proper initialization. The contracts that will serve as implementations for Factories also should be properly initialized, to prevent the undesired state modifications on it.

7.2 DefaultAllocationProxy Allocate

Note Version 1

The design of `allocate` function in DefaultAllocationProxy contract has aspect that is worth mentioning: The failure of any beneficiary fallback logic will force the entire allocation procedure to fail. In current implementation version only the QHolderRewardProxy has any logic in its receive function.

7.3 Plain Strings as Keys

Note Version 1

The use of plain strings as access keys for the registry across the code base is error prone. It could lead to mistyping one of the keys, can make a contract unusable. A less error prone solution would be to store those string keys as globally available constants.

7.4 Proxy Tests

Note Version 1

The truffle tests concerning proxies should be extended. In the current state, the proxies are just deployed, but never used. Tests should be done with real proxies, issues like the one concerning the ownership could have been detected then.

7.5 Roots Function _addStake Check

Note Version 1

The check needs to check the `_amount` value. In current implementation this causes no problems, but can lead to bug if functionality changes.

```
require(msg.value > 0, "[QEC-002012]-Additional stake must not be zero.");
```

7.6 SlashingEscrow `PENDING` After `appealEndTime`

Note Version 1

The `ASlashingEscrow`'s state machine can stays `PENDING` on an arbitration decision if not enough `RootNodes` confirm the decision. If a decision on a casted objection does not receive enough confirmations, it stays on `PENDING` state, event after the `appealEndTime`. So as long as not enough `RootNodes` have confirmed the decision, the slashed amount is kept in the slashing escrow, that could mean pure loss for the validator, the slashing proposer and the system reserve if the arbitration is never decided.

The Q Blockchain team confirmed, that root nodes are incentivized to vote. Thus every vote on `SlashingEscrow` should eventually be decided.

7.7 System Compatibility With External Tokens

Note Version 1

While systems contracts operate properly with good behaving ERC20 tokens, some odd tokens can cause potential problems in the system. For example, the `BorrowingCore` getting collateral with `collateral.transferFrom(msg.sender, address(this), _amount)` and `_amount` is retained for accounting. This behavior does not tolerate tokens with fees. If a collateral is a token that allows fees upon transfer, the real amount received by the BorrowingCore will be less than `_amount` and the system could end up undercollateralized without noticing it. Similarly, the rounding errors in QVault transfers can lead to smaller received values on the BorrowingCore side. Thus, QVault cannot be used as a collateral inside BorrowingCore without code adjustments. In addition, on Ethereum some tokens don't always return values on transfer/transferFrom or approvals. Some tokens have unusual number of decimals (too big or too small). Allowing the system to manipulate any external token should be done with a great care.

7.8 Total Q Supply Assumption Source

Note Version 1

Upon the computation of total circulating `Q` amount, `(block.number.mul(15)).add(10000000000).mul(1 ether)` is copy-pasted as-is four times. Having system parameters defined in different sources is error prone and can complicate the upgradability of the system contracts.

7.9 Unchecked Function Return Values

Note Version 1

Some of the calls to known functions are not checked, mainly calls to `AddressStorage`.

Examples :

- `QVault.delegateStake` : return value of `_newDelegations.add(_delegationAddr);` is not checked

- `AExertsMembership.add/remove/swapMember` : return values of the interactions with the list of experts are not checked
- `QVault` `_addToBalance` and `_subFromBalance` functions return bool that are never used

While this might be intended, this uses more gas.