SMART CONTRACT AUDIT REPORT

for

QIAN STABLECOIN PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China

August 24, 2020

Document Properties

Client	QIAN Stablecoin Governance Committee
Title	Smart Contract Audit Report
Target	QIAN 2.0
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Aug. 24, 2020	Xuxian Jiang	Final Release
1.0-rc1	Aug. 20, 2020	Xuxian Jiang	Release Candidate #1
0.3	Aug. 19, 2020	Xuxian Jiang	Additional Findings #2
0.2	Aug. 15, 2020	Xuxian Jiang	Additional Findings #1
0.1	Aug. 12, 2020	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **QIAN 2.0** smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About QIAN 2.0

QIAN 2.0 is a decentralized stablecoin ecosystem where everyone can equally, freely and conveniently participate and enjoy non-discriminatory financial services. Similar to existing stablecoin offerings, QIAN requires collateralized cryto-assets to back up the value and maintain 1:1 parity with different fiat currencies such as USD (The collateral is then locked up in a smart contract). However, unlike other stablecoin solutions, QIAN 2.0 does not charge any interest fee with the intended goal of greatly broadening the adoption base and embracing a variety of usage scenarios.

The basic information of QIAN 2.0 is as follows:

Table 1.1: Basic Information of QIAN 2.0

Item	Description
Issuer	QIAN Stablecoin Governance Committee
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 24, 2020

In the following, we show the repository of reviewed code used in this audit.

- https://github.com/QIAN-Protocol/QIAN (303515c)

1.2 About PeckShield

PeckShield Inc. [24] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [19]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [18], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as an investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the QIAN 2.0 implementation. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	3	■■■
Medium	4	■■■■
Low	7	■■■■■■■
Informational	2	■■
Total	16	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 high-severity vulnerabilities, 4 medium-severity vulnerabilities, 7 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key QIAN 2.0 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Necessity of Single-Shot Initialization	Init. and Cleanup	Fixed
PVE-002	Low	Votability of Executed/Dropped Proposals	Business Logics	Fixed
PVE-003	Low	Open Activation of Deployed Proposals	Business Logics	Fixed
PVE-004	High	Overly-Privileged Governance Auditors	Security Features	Confirmed
PVE-005	Medium	Improved Sanity Checks For Upgrade	Error Conditions, Return Values, Status Codes	Fixed
PVE-006	Low	Missing Information in Upgrade Events	Error Conditions, Return Values, Status Codes	Fixed
PVE-007	Info.	Mis-handled Corner Cases in CSA State Classification	Business Logics	Fixed
PVE-008	Medium	Lack of Sanity Checks in setades()	Error Conditions, Return Values, Status Codes	Fixed
PVE-009	High	Lack of Global Adequacy Ratio Enforcement	Security Features	Fixed
PVE-010	High	Removed Tokens For Stablecoin Minting	Business Logics	Fixed
PVE-011	Info.	Unimplemented Liquidation Redemption Factor	Business Logics	Confirmed
PVE-012	Low	Incompatibility with Deflationary Tokens	Business Logics	Fixed
PVE-013	Medium	Tightened Access Controls Between Main And Modules	Coding Practices	Fixed
PVE-014	Low	Corner Case Handling in Rate Assessment	Numeric Errors	Fixed
PVE-015	Low	approve()/transferFrom() Race Condition	Time and State	Fixed
PVE-016	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Necessity of Single-Shot Initialization

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Authenticable.sol`
- Category: Initialization and Cleanup [16]
- CWE subcategory: CWE-1188 [3]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize`) that basically executes all the setup logic.

However, a follow-up caveat is that during a contract's lifetime, its constructor is guaranteed to be called exactly once (and it happens at the very moment of being deployed). But a regular function may be called multiple times! In order to ensure that a contract will only be initialized once, we need to guarantee that the chosen `initialize` function can be called only once during the entire lifetime. This guarantee is typically implemented as a modifier named `initializer`.

QIAN 2.0 implements the upgradeability logic in `UpgradeableDelegatecallFallback` and provides the `initializer` modifier support in `Authenticable`. To facilitate our discussion, we show the code snippet of `initializer` below.

```
    modifier initializer() virtual {
        require(
```

```
            authenticable() != address(0),
            "Authenticable.auth/authenticable uninitialized"
        );
        require(
            IAuthentication(authenticable()).accessible(
                msg.sender,
                address(this),
                msg.sig
            ),
            "Authenticable.auth/operation unauthorized"
        );
        _;
    }
```

Listing 3.1: Authenticable.sol

Apparently the above logic only protects the caller is authenticated and allowed by the system. But it does not provide the guarantee that the `initialize` function attached with the `initializer` modifier can be called only once. Considering the need of multiple versions arranged for future upgrades, we strongly suggest the adoption of the known `VersionedInitializable` implementation.

Recommendation Adopt the `VersionedInitializable` contract for proper initialization with the required guarantee of executing the intended `initialize` function only once during the entire lifetime.

```
    /**
     * @dev Modifier to use in the initializer function of a contract.
     */
    modifier initializer() {
        uint256 revision = getRevision();
        require(
            initializing
                isConstructor()
                revision > lastInitializedRevision,
            "Contract instance has already been initialized"
        );

        bool isTopLevelCall = !initializing;
        if (isTopLevelCall) {
            initializing = true;
            lastInitializedRevision = revision;
        }

        _;

        if (isTopLevelCall) {
            initializing = false;
        }
    }
```

Listing 3.2: Authenticable.sol

3.2 Votability of Executed/Dropped Proposals

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Proposal.sol
- Category: Business Logics [13]
- CWE subcategory: CWE-841 [10]

Description

QIAN 2.0 defines a standard work-flow to submit, vote, and execute proposals that enact on the system-wide operations and upgrades. A proposal falls in four different states, i.e., `None`, `ACTIVATED`, `EXECUTED`, and `REVOKED`. The `None` state means that the proposal is not valid or has not been submitted; the `ACTIVATED` state indicates that the proposal has been activated and is now open for votes; the `EXECUTED` state shows the proposal has been selected and successfully executed; and the `REVOKED` states cancels the proposal execution.

Our analysis shows that the `vote()` function handles the user votes and selects current top candidate with most votes. The logic is rather straightforward. However, we notice there are two specific requirements before entering the voting process: `require(actv == 1, "Resolution.vote/vote unactivated")` and `require(exp > now, "Resolution.vote/vote expired")`. The first one ensures the proposal has been `ACTIVATED` and thus is open for votes. The second one guarantees the proposal has not expired yet. Unfortunately, it does not check whether the proposal has been `EXECUTED` or `REVOKED`. In other words, an already executed (or revoked) proposal is still open for votes, which may essentially lock up the assets staked with the vote for a certain duration. Though the voter can later reclaim back the staked assets, the lock-up period implies unnecessary loss of associated opportunity cost.

```
    function vote(address candidate, uint256 amount) public {
        //TODO: TEST candidate
        require(actv == 1, "Resolution.vote/vote unactivated");
        require(exp > now, "Resolution.vote/vote expired");
        require(
            voters[msg.sender].candidate == address(0) ||
                voters[msg.sender].candidate == candidate,
            "Resolution.vote/vote immutable"
        );

        balances[msg.sender] = balances[msg.sender].add(amount);
        voters[msg.sender].candidate = candidate;
        voters[msg.sender].vote = voters[msg.sender].vote.add(amount);

        votes[candidate] = votes[candidate].add(amount);
        accvotes = accvotes.add(amount);
```

```
        if (votes[candidate] >= votes[top]) {
            top = candidate;
        }

        IERC20(tok).safeTransferFrom(msg.sender, address(this), amount);
        emit Vote(msg.sender, candidate, amount);
    }
```

Listing 3.3: Proposal.sol

Recommendation Ensure that an executed (or dropped) proposal cannot take any new vote.

```
    function vote(address candidate, uint256 amount) public {
        //TODO: TEST candidate
        require(actv == 1, "Resolution.vote/vote unactivated");
        require(done == 0 && drop==0, "Resolution.vote/vote executed or dropped")
        require(exp > now, "Resolution.vote/vote expired");
        require(
            voters[msg.sender].candidate == address(0)
                voters[msg.sender].candidate == candidate,
            "Resolution.vote/vote immutable"
        );

        balances[msg.sender] = balances[msg.sender].add(amount);
        voters[msg.sender].candidate = candidate;
        voters[msg.sender].vote = voters[msg.sender].vote.add(amount);

        votes[candidate] = votes[candidate].add(amount);
        accvotes = accvotes.add(amount);

        if (votes[candidate] >= votes[top]) {
            top = candidate;
        }

        IERC20(tok).safeTransferFrom(msg.sender, address(this), amount);
        emit Vote(msg.sender, candidate, amount);
    }
```

Listing 3.4: Proposal.sol

3.3 Open Activation of Deployed Proposals

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Proposal.sol`
- Category: Business Logics [13]
- CWE subcategory: CWE-841 [10]

Description

As described in Section 3.1, QIAN 2.0 defines a standard work-flow to submit, vote, and execute proposals that enact on the system-wide operations and upgrades. A proposal falls in four different states, i.e., `None`, `ACTIVATED`, `EXECUTED`, and `REVOKED`. Our analysis shows that the `activate()` function implements the logic to officially submit a new proposal into the system.

We notice that this `activate()` function is defined as `public` and there is no access control policy guarding its access, rendering this function open for any one to call. Consequently, a deployed proposal may expose a time window of vulnerability that can be exploited by any one to activate it. After the activation, the proposal will not accept others to `execute` or `revoke` it, rendering the proposal useless from the governance perspective. As a result, the governance cannot enact on any proposal activated by others!

```
    function activate() public {
        require(
            authentication == address(0),
            "Resolution.activate/activated authentication"
        );
        authentication = msg.sender;
        pend = 1;
    }
```

Listing 3.5: Proposal.sol

Recommendation Place an access control modifier with `activate()` so that it can only be activated by the governance contract. An example implementation is shown as follows:

```
    modifier onlyGov() {
        require(msg.sender == governor, "operation unauthorized");
        _;
    }

    function activate() public onlyGov {
        require(status == NONE, "Proposal.activate/unexpected status");
        status = ACTIVATED;
    }
```

Listing 3.6: Proposal.sol

3.4 Overly-Privileged Governance Auditors

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Governance.sol`
- Category: Security Features [11]
- CWE subcategory: CWE-287 [5]

Description

In QIAN 2.0, the `Governance` contract plays a critical role in governing and regulating the system-wide operations (e.g., configuration and upgradeability). It also has the privilege to fully control or govern the life-cycle of proposals and enact them regarding their submissions, executions, and revocations.

With great privilege comes great responsibility. Our analysis shows that governance contract is indeed privileged, but it should NOT empower the associated governance auditor (using the same terminology from QIAN 2.0 code base) to become omnipotent! In particular, by leveraging and exploiting the `Governance` contract's role, an auditor can essentially have the capability of immediately activating and executing any customized proposal. The proposal can be tasked with configuring any system-wide risk parameters (e.g., a token's frozen adequacy ratio) or even moving all assets out of the vault!

```
    modifier auditauth {
        require(
            auditable(msg.sender),
            "Governance.auditauth/operation unauthorized"
        );
        _;
    }

    //
    function activate(address proposal) public auditauth {
        require(proposals[proposal] == NONE, "Governance.execute/reactivated");
        IProposal(proposal).activate();
        proposals[proposal] = ACTIVATED;
        emit Activate(msg.sender, proposal);
    }

    //
    function execute(address proposal) public auditauth returns (bytes memory) {
        require(
            proposals[proposal] == ACTIVATED,
            "Governance.execute/unactivated"
        );
        _weights[proposal] = 1;
        (bool success, bytes memory result) = IProposal(proposal).execute();
        require(success, "Governance.execute/failed");
```

```
71          _weights[proposal] = 0;
72          proposals[proposal] = EXECUTED;
73          emit Execute(msg.sender, proposal);
74          return result;
75      }
```

Listing 3.7: Governance.sol

In particular, if we examine the above code snippets, we realize that any `Governance` auditor can craft a proposal (e.g., with `exp=0`) that can be immediately activated via the `Governance`'s `activate()`. After that, the auditor can immediately invoke `execute()` that elevates the proposal privilege by assigning `_weights[proposal] = 1` (line 68). And these steps can be completed within a single transaction! With the elevated privilege, the proposal is essentially granted to access and configure various aspects of the system, including unexpected withdrawal of all assets in the vault.

The current overly-privileged design of governance auditors makes this system not compatible to the usual trustless setup for reduced risks or shared responsibilities. A typical path may begin with a centralized governance in the early, formative days and gradually shifts over time to a community-based governance system. The system does have a community-based governance mechanism in place and can dynamically configure a new auditor with the approvals of a majority of the current independent parties. We just need to make a step further to restrict the auditors and effectively prevent them from abusing their granted privilege.

Recommendation Contain the privileges of `Governance` auditors. In the meantime, develop a long-time plan for eventual community-based governance and ensure the intended trustless nature and high-quality distributed governance.

3.5 Improved Sanity Checks For Upgrade

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `UpgradeableDelegatecallFallback`
- Category: Status Codes [15]
- CWE subcategory: CWE-391 [7]

Description

As mentioned in Section 3.1, there is a need for upgradeable contract design and the upgradeability support comes with a few caveats. Besides the previous caveats, a new caveat comes along with the mixed upgradeability and authentication.

In the following, we show the code snippet of current implementation. The latest implementation contract is recorded in the proxy's storage slot `IMPLEMENTATION_STORAGE_SLOT`. And its modification is guarded with an access control modifier `auth`.

```
26     function upgrade(address _implementation, bytes memory _data)
27         public
28         payable
29         auth
30     {
31         require(
32             _implementation != address(0),
33             "Upgradable.upgrade/unexpected implementation"
34         );
35         _initializeimplementation(_implementation);
36         if (_data.length > 0) {
37             (bool success, ) = _implementation.delegatecall(_data);
38             require(success, "Upgradable.upgrade/initialization aborted");
39         }
40         emit Upgrade(msg.sender, authenticable(), _implementation, _data);
41     }

43     function upgrade(
44         address _authentication,
45         address _implementation,
46         bytes memory _data
47     ) public payable auth {
48         _initializeauthenticable(_authentication);
49         upgrade(_implementation, _data);
50         emit Upgrade(msg.sender, _authentication, _authentication, _data);
51     }
```

Listing 3.8: UpgradeableDelegatecallFallback.sol

```
    modifier auth() virtual {
        require(
            authenticable() != address(0),
            "Authenticable.auth/authenticable uninitialized"
        );
        require(
            IAuthentication(authenticable()).accessible(
                msg.sender,
                address(this),
                msg.sig
            ),
            "Authenticable.auth/operation unauthorized"
        );
        _;
    }
```

Listing 3.9: Authenticable.sol

Specifically, the `upgrade()` function allows any caller with necessary authorization to upgrade current implementation and authentication logic (lines 48 – 49). While the implementation update has been properly guarded to ensure the new implementation will not be reset (by assigning with address(0)), the authentication update logic may reset the authentication to be address(0). A zeroed authentication can accidentally disable the access to all auth-protected interfaces and necessitate immediate actions for fix-up!

Recommendation Apply additional sanity checks in the upgrade() routine so that the new _authentication can not be zero! Also, it is suggested to remove redundant check inside the same routine, i.e., _implementation != address(0), since it is always evaluated true.

3.6 Missing Information in Upgrade Events

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: UpgradeableDelegatecallFallback
- Category: Error Conditions [15]
- CWE subcategory: CWE-391 [7]

Description

There exist another issue regarding the same upgrade logic we discussed in Section 3.5. We notice there are two upgrade() routines with different parameters. The first one performs the actual update to the new implementation and the second one encapsulates the first one with the additional functionality to update new authentication as well.

We point out that the first upgrade() routine emits an event Upgrade(msg.sender, authenticable(), _implementation, _data) and the second upgrade() emits Upgrade(msg.sender, _authentication, _authentication, _data). The definition of this particular event is as follows: event Upgrade(address sender, address authentication, address implementation, bytes data).

```
    function upgrade(address _implementation, bytes memory _data)
        public
        payable
        auth
    {
        require(
            _implementation != address(0),
            "Upgradable.upgrade/unexpected implementation"
        );
        _initializeimplementation(_implementation);
        if (_data.length > 0) {
            (bool success, ) = _implementation.delegatecall(_data);
            require(success, "Upgradable.upgrade/initialization aborted");
        }
        emit Upgrade(msg.sender, authenticable(), _implementation, _data);
```

```
    }

    function upgrade(
        address _authentication,
        address _implementation,
        bytes memory _data
    ) public payable auth {
        _initializeauthenticable(_authentication);
        upgrade(_implementation, _data);
        emit Upgrade(msg.sender, _authentication, _authentication, _data);
    }
```

Listing 3.10: UpgradeableDelegatecallFallback.sol

Apparently, the second event is emitted by mistakenly including `_authentication` as `_implementation`. Also, there exists unnecessary redundancy in emitting basically identical events twice. Since the latest implementation information is critical for the entire protocol operation, we believe there is an absolute need to ensure its accuracy and freshness.

Recommendation Revise the above event generation by removing unnecessary redundancy and providing proper information.

3.7 Mis-handled Corner Cases in CSA State Classification

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Main`
- Category: Business Logics [13]
- CWE subcategory: N/A

Description

According to the white paper of QIAN 2.0, the CSA may be classified into three categories: `CSA(normal)`, `CSA(alarm)`, and `CSA(frozen)`. The first category `CSA(normal)` essentially shows a healthy fully-collateralized position with more than 200% adequacy ratio, i.e., $Y > 200\%$ (measured by the collateral/supply ratio); the second category `CSA(alarm)` indicates the ratio between 120% and 200%, or more precisely, $120\% < Y <= 200\%$; the last category `CSA(frozen)` represents a risky position, $Y <= 120\%$, and the CSA assets are frozen for liquidation.

If we examine the implementation logic to classify the CSAs, there is a slight discrepancy. Specifically, the first category is determined when `ade(owner, token)>= IENV(env).bade(token)`, i.e., $Y >= 200\%$ (not $Y > 200\%$). In addition, the last category is considered when `ade(owner, token)< IENV(env).fade(token)`, i.e., $Y < 120\%$ (not $Y <= 120\%$). Apparently, the boundary cases are inconsistent with the white paper.

```
    //
    function _isbade(address owner, address token)
        internal
        view
        returns (bool)
    {
        return ade(owner, token) >= IENV(env).bade(token);
    }

    //
    function _isfade(address owner, address token)
        internal
        view
        returns (bool)
    {
        return ade(owner, token) < IENV(env).fade(token);
    }
```

Listing 3.11: Main.sol

Recommendation Make the classification consistent between the white paper and the actual implementation.

3.8 Lack of Sanity Checks in setades()

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `ENV.sol`
- Category: Status Codes [15]
- CWE subcategory: CWE-391 [7]

Description

QIAN 2.0 has a built-in `ENV` contract that allows the specification of various system-wide risk parameters, including the adequacy ratio for each supported asset and the cap that limits the total mint-able amount of `QIAN` stable coins. Evidently, these risk parameters ensure proper protocol execution and require great care in their changes or customizations.

In the following, we show the code snippet of `setades()` that updates customizable risk parameters for each supported token, i.e., `bade`, `aade`, and `fade`. The first parameter `bade` is the threshold above which the CSA can be classified as `CSA(normal)`; The third parameter `fade` controls the threshold below which the CSA can be classified as `CSA(frozen)`.

```
    function setades(uint256 nades, bytes[] memory ades) public auth {
        require(nades == ades.length, "Environment.setades/msimatch arguments");
        for (uint256 i = 0; i < nades; ++i) {
```

```
            (address _token, uint256 _bade, uint256 _aade, uint256 _fade) = abi
                .decode(ades[i], (address, uint256, uint256, uint256));
            swapenvs[_token].bade = _bade;
            swapenvs[_token].aade = _aade;
            swapenvs[_token].fade = _fade;
        }
    }
```

Listing 3.12: ENV.sol

As this routine updates these important parameters that may impact the overall operation and healthiness, great care needs to be taken to ensure these parameters fall in normal range. Currently, there is no sanity checks in place to ensure their correctness.

In addition, as these parameters control various aspects of system operation, there is a need to emit related events accordingly.

Recommendation Apply necessary sanity checks to ensure these parameters always fall in proper range. Also emit corresponding events when these risk parameters are being updated.

```
    event ADE(address indexed sender, address indexed token, uint bade, uint aade, uint
        fade);

    function setades(uint256 nades, bytes[] memory ades) public auth {
        require(nades == ades.length, "Environment.setades/msimatch arguments");
        for (uint256 i = 0; i < nades; ++i) {
            (address _token, uint256 _bade, uint256 _aade, uint256 _fade) = abi
                .decode(ades[i], (address, uint256, uint256, uint256));
            swapenvs[_token].bade = _bade;
            swapenvs[_token].aade = _aade;
            swapenvs[_token].fade = _fade;

            emit ADE(msg.sender, _token, _bade, _aade, _fade);
        }

        require(_bade > _aade && _aade > _fade && _bade >= 2*1e18 && _fade >= 1e18
                "Environment.setades/unexpected parameters"
        );
    }
```

Listing 3.13: ENV.sol (revised)

3.9 Lack of Global Adequacy Ratio Enforcement

- ID: PVE-009
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Main.sol`
- Category: Security Features [11]
- CWE subcategory: CWE-287 [5]

Description

As discussed in Section 3.8, QIAN 2.0 has a built-in `ENV` contract that allows the specification of various system-wide risk parameters. One particular parameter is the global adequacy ratio `gade`, which in essence defines the overall system healthiness. If properly enforced, the parameter can guarantee that the entire system will not run into a default state. Apparently, these risk parameters under-pin proper protocol execution and require great care in their changes or customizations.

In the following, we show the code snippet of `setgade()` that updates the global adequacy ratio. However, this global risk parameter is not enforced at all.

```
    function setgade(uint256 _gade) public auth {
        gade = _gade;
    }
```

Listing 3.14: ENV.sol

For example, if we take a look at the QIAN-minting logic (implemented in `mint` as shown below), it properly ensures the minimum amount `step` (line 323), guarantees the healthiness of affected CSA, and falls below the allowed `line` limit. However, it fails to check the global adequacy ratio `gade` even though new minting behavior will reduce it.

```
321     function _mint(address token, uint256 supply) internal {
322         uint256 _step = IENV(env).step();
323         require(supply >= _step, "Main.mint/mismatch minimum-supply");

325         IMintable(coin).mint(msg.sender, supply);
326         IBalance(balance).mint(msg.sender, token, supply);

328         require(_isbade(msg.sender, token), "Main.mint/insufficient reserve");
329         uint256 _supply = IBalance(balance).supply(token);
330         uint256 _line = IENV(env).line(token);
331         require(_supply <= _line, "Main.mint/mismatch maximum-supply");

333         IRate(rate).onmint(msg.sender, supply);
334     }
```

Listing 3.15: Main.sol

In addition to `mint()`, there are a few others routines that can also affect the global adequacy ratio and thus need to be properly hardened for the enforcement. These routines include `withdraw()`, `open()`, and `exchange()`. The revision in `withdraw()` may require additional care in not blocking legitimate, non-frozen CSA holders from withdrawing their assets.

Recommendation Apply necessary sanity checks to ensure the global adequacy ratio will not be violated. In the following, we show one example enforcement in `mint()`. Note the enforcement is placed at the end after the minting is done, instead of at the beginning before the minting.

```
    function _mint(address token, uint256 supply) internal {
        uint256 _step = IENV(env).step();
        require(supply >= _step, "Main.mint/mismatch minimum-supply");

        IMintable(coin).mint(msg.sender, supply);
        IBalance(balance).mint(msg.sender, token, supply);

        require(_isbade(msg.sender, token), "Main.mint/insufficient reserve");
        uint256 _supply = IBalance(balance).supply(token);
        uint256 _line = IENV(env).line(token);
        require(_supply <= _line, "Main.mint/mismatch maximum-supply");
        require(_isgade(), "Main.mint/risky gade");

        IRate(rate).onmint(msg.sender, supply);
    }
```

Listing 3.16: Main.sol (revised)

3.10 Removed Tokens For Stablecoin Minting

- ID: PVE-010
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Main`
- Category: Business Logics [13]
- CWE subcategory: CWE-754 [9]

Description

The design of QIAN 2.0 allows for dynamic addition and removal of chosen tokens as collaterals for stablecoin minting. This process typically subjects to a formal governance procedure that requires the submission of an associated proposal. The proposal will then be open for public votes. Once passed, the proposal will then be enacted to complete the addition or removal of the specified token.

```
    function removetoken(address token) public auth {
        require(_tokens.contains(token), "env.enabletoken/token is not exists");
        _tokens.remove(token);
```

```
    }
```

Listing 3.17: ENV.sol

Our analysis of the removal logic indicates that it indeed removes the token from the internal array of supported tokens. However, the minting process is not updated and still allows the removed tokens to be used as collaterals for stablecoin minting. We believe it is necessary to apply rigorous validity checks to prevent removed tokens from being used as collateralized assets for stablecoin minting.

```
    function _mint(address token, uint256 supply) internal {
        uint256 _step = IENV(env).step();
        require(supply >= _step, "Main.mint/mismatch minimum-supply");

        IMintable(coin).mint(msg.sender, supply);
        IBalance(balance).mint(msg.sender, token, supply);

        require(_isbade(msg.sender, token), "Main.mint/insufficient reserve");
        uint256 _supply = IBalance(balance).supply(token);
        uint256 _line = IENV(env).line(token);
        require(_supply <= _line, "Main.mint/mismatch maximum-supply");

        IRate(rate).onmint(msg.sender, supply);
    }
```

Listing 3.18: Main.sol

Recommendation Removed tokens should be blocked from being able to mint stablecoins. In order to achieve that, we need to apply necessary sanity checks to prevent a removed token from entering `_mint()`.

```
    function mint(address token, uint256 supply) public nonReentrant {
        require(IENV(env).hastoken(token) && !IENV(env).deprecatedtoken(token),
            "Main.mint/unexpected token"
        );
        _mint(token, supply);
        emit Mint(msg.sender, token, supply, IERC20(coin).totalSupply());
    }
```

Listing 3.19: Main.sol

3.11 Unimplemented Liquidation Redemption Factor

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Main`
- Category: Business Logics [13]
- CWE subcategory: N/A

Description

As discussed in Section 3.7, the white paper of QIAN 2.0 elaborates the purpose of CSAs and classifies them into three categories: `CSA(normal)`, `CSA(alarm)`, and `CSA(frozen)`. The collateralized assets in `CSA(frozen)` are open for liquidation. The white paper also specifies a so-called liquidation redemption percentage or factor α that is designed to better protect frozen CSA holders by allowing only the specified percentage of collateralized assets (that belong to a frozen CSA) for liquidation.

We have examined the liquidation logic implemented in the `exchange()` routine. Our analysis shows that the redemption factor α has not kicked in the process as all assets in `CSA(frozen)` can be liquidated. This inconsistency between smart contracts and the white paper is noted and needs to be resolved in either implementing the full logic or revising the white paper to ensure consistency.

```
    function exchange(
        uint256 supply, //QIAN
        address token,
        address[] memory frozens
    ) public nonReentrant {
        require(_isgade(), "Main.mint/risky gade");
        require(IENV(env).risk(token) == 0, "Main.exchange/risky token");
        require(supply != 0, "Main.exchange/unexpected exchange supply");
        address[] memory _frozens = _refreshfrozens(token, frozens);
        require(_frozens.length != 0, "Main.exchange/no frozens");

        //fix:
        IBalance.swap_t[] memory swaps = new IBalance.swap_t[](_frozens.length);
        for (uint256 i = 0; i < _frozens.length; ++i) {
            //fix: Stack too deep, try removing local variables.
            (address _owner, address _token) = (_frozens[i], token);
            swaps[i] = IBalance(balance).swaps(_owner, _token);
        }

        uint256 _supply = supply;
        uint256 reserve = 0;
        for (uint256 i = 0; i < _frozens.length; ++i) {
            //fix: Stack too deep, try removing local variables.
            (address _owner, address _token) = (_frozens[i], token);

            uint256 rid = _min(swaps[i].supply, _supply);
```

```
            _supply = _supply.sub(rid);

            uint256 lot = rid.mul(swaps[i].reserve).div(swaps[i].supply);
            lot = _min(lot, swaps[i].reserve);

            IBalance(balance).exchange(msg.sender, _owner, _token, rid, lot);
            IRate(rate).onburn(_owner, rid);

            reserve = reserve.add(lot);

            if (_supply == 0) break;
        }

        uint256 __supply = supply.sub(_supply);
        IBurnable(coin).burn(msg.sender, __supply);

        _withdraw(token, reserve);
        emit Exchange(
            msg.sender,
            __supply,
            token,
            reserve,
            IAsset(asset).balances(token),
            IERC20(coin).totalSupply(),
            _frozens,
            _price(token)
        );
    }
```

Listing 3.20: Main.sol

The analysis of the above logic also indicates that the liquidation incentive could be as high as $ENV.fade(token) - 1$. If we assume an example setting of frozen adequacy ratio, i.e., $ENV.fade(token)$, is 120%, the liquidation incentive can be as high as 20%. Such high incentive is achieved at the cost of frozen CSA holders, therefore discouraging their participation. We feel strongly the need to explore further trade-offs of establishing an appropriate liquidation incentive. As our suggestion, an alternative will be to set up a new risk parameter that can be dynamically adjusted via the governance process.

Recommendation Be consistent between the design document and the actual implementation. Also consider the addition of a new risk parameter for governance-regulated dynamic liquidation incentive adjustment.

3.12 Incompatibility with Deflationary Tokens

- ID: PVE-012
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Main`, `Proposal`
- Category: Business Logics [13]
- CWE subcategory: CWE-841 [10]

Description

In QIAN 2.0, the `Main` contract is designed to be the interface and interact with users. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract provides low-level routines to transfer assets into or out of its vault, i.e., the `asset` contract (see the code snippet below). These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(address token, uint256 reserve)
        public
        payable
        nonReentrant
    {
        IAsset(asset).deposit.value(msg.value)(msg.sender, token, reserve);
        IBalance(balance).deposit(msg.sender, token, reserve);
        ISids(sids).push(msg.sender, token);
        emit Deposit(msg.sender, token, reserve, IAsset(asset).balances(token));
    }

    function withdraw(address token, uint256 reserve) public nonReentrant {
        _withdraw(token, reserve);
        require(!_isfade(msg.sender, token), "Main.withdraw/frozen");
        emit Withdraw(
            msg.sender,
            token,
            reserve,
            IAsset(asset).balances(token)
        );
    }
```

Listing 3.21: Main.sol

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer or transferFrom. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as deposit() and withdraw(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of target and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in transfer or transferFrom will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the transfer or transferFrom is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into QIAN 2.0 for indexing. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the transferFrom() call to ensure the book-keeping amount is accurate. The affected smart contracts include Main and Proposal.

An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

3.13 Tightened Access Controls Between Main And Modules

- ID: PVE-013
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Main.sol
- Category: Security Features [11]
- CWE subcategory: CWE-287 [5]

Description

In QIAN 2.0, the Main contract is the hub that connects various modules, including Asset, Balance, ENV, Fund, QIAN, Rate, and Sids. The addresses of these modules are naturally passed when the Main contract is being initialized.

```
    function initialize(
        address _env,
        address _balance,
        address _asset,
        address _sids,
        address _coin,
        address _rate,
```

```
        address _fund
    ) public initializer {
        ReentrancyGuard.initialize();

        env = _env;
        balance = _balance;
        asset = _asset;
        sids = _sids;
        coin = _coin;
        rate = _rate;
        fund = _fund;
    }
```

Listing 3.22: Main.sol

To seamlessly integrate these modules, QIAN 2.0 provides a powerful authentication chain that allows for flexible, fine-grained authorization and enforcement for each privileged interface. For example, the `QIAN` token is minted (or redeemed) via the `mint()` (or `burn()`) interface; the `asset` vault is only accessible via `deposit()/withdraw()`; the internal accounting, i.e., `balance`, can only be updated via `deposit()/withdraw()`; the `rate` updates are performed via `onmint()/onburn()/ongain()`. These interfaces are privileged and hence guarded with the `auth` modifier.

```
25    modifier auth() virtual {
26        require(
27            msg.sender == address(this) ||
28                IAuthentication(authenticable()).accessible(
29                    msg.sender,
30                    address(this),
31                    msg.sig
32                ),
33            "Authenticable.auth/operation unauthorized"
34        );
35        _;
36    }
```

Listing 3.23: Authenticable.sol

Our analysis shows that many of these privileged interfaces are supposed to be invoked by the `Main` contract only, despite the development of a rather sophisticated authentication chain (through `accessible` in line 28). The authentication chain, though powerful, may unnecessarily make the protocol management and operation complicated. Any complicated operation may be error-prone.

In the meantime, since these interfaces should be called only by a known trusted entity, i.e., `Main`, it is strongly suggested to codify such restriction in the smart contract logic to avoid any unintended human error (when configuring and managing the authentication chain). By doing so, we also properly follow the best practice of granting least privilege principle in minimizing possible attack vectors.

Recommendation Tighten the access control policy on the above-mentioned modules so that they can only be accessed from `Main` (e.g., by introducing a new modifier `onlyMain`).

3.14 Corner Case Handling in Rate Assessment

- ID: PVE-014
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Rate`
- Category: Numeric Errors [17]
- CWE subcategory: CWE-190 [4]

Description

QIAN 2.0 provides an interesting feature that incentives CSA holders for their stablecoin minting. The scale of incentive is controlled by a system-wide risk parameter `ENV.growth()`. This parameter will materialize actual gains for CSA holders.

The incentive logic is implemented in `Rate` and is accessed through a few hooks, i.e., `onmint()`, `onburn()`, and `ongain()`. These hooks are supposed to be invoked when stablecoins are minted or redeemed. Inside these hooks, we notice the gain calculation has a specific requirement, i.e., `require(F <= 2**255)`.

Before delving into the details, we use the `onburn()` hook as an example. Its logic basically calculates the incentive amount the CSA holder is supposed to be rewarded due to the previously-minted amount, i.e., `supply`. (And this amount is currently being redeemed.) Internally, the `Rate` contract maintains a data structure `reward_t` for each CSA holder and this structure has a member, i.e., `int256 gain`, that actually records the holder's reward amount.

```
    //
    function onmint(address who, uint256 supply) public auth {
        //
        uint256 F = supply.mul(IENV(envs).growth()).mul(now).div(1E18);
        require(F <= 2**255, "Rate.onmint/unexpected overflow");
        rewards[who].gain = rewards[who].gain.add(-int256(F));
        rewards[who].supply = rewards[who].supply.add(supply);
    }

    //
    function onburn(address who, uint256 supply) public auth {
        //
        uint256 F = supply.mul(IENV(envs).growth()).mul(now).div(1E18);
        require(F <= 2**255, "Rate.onburn/unexpected overflow");
        rewards[who].gain = rewards[who].gain.add(int256(F));
        rewards[who].supply = rewards[who].supply.sub(supply);
    }

    //
```

```
    function ongain(address who, uint256 amount) public auth {
        require(amount <= 2**255, "Rate.ongain/unexpected overflow");
        require(gains(who) >= amount, "Rate.ongain/insufficient gains");
        rewards[who].gain = rewards[who].gain.add(-int256(amount));
    }
```

Listing 3.24: Rate.sol

It is noteworthy that for signed integer, the maximum `int256` number is $max\_int256 = 2 ** 255 - 1$, while the minimum `int256` is $min\_int256 = -2 ** 255$. Recall the previous requirement `require(F <= 2**255)`. This requirement basically limits the ceiling of `F`, i.e., $max\_int256 + 1$, which means the ceiling, if reached, will overflow and become min_int256. In other words, $int256(2 ** 255) = -2 ** 255 = min\_int256$.

We emphasize this corner case is unlikely to occur. However, to avoid unnecessary implications, since ceiling of `F` is not reached, we suggest to require `F` to be strictly less than $2 ** 255$, i.e., `require(F < 2**255)`.

Recommendation Revise these requirements to avoid unnecessary signed integer overflow during subsequent type conversion.

3.15 approve()/transferFrom() Race Condition

- ID: PVE-015
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `QIAN`
- Category: Time and State [12]
- CWE subcategory: CWE-362 [6]

Description

`QIAN` is an ERC20 token that represents the stablecoins that can be minted or redeemed. In current implementation, there is a known race condition issue regarding `approve()` / `transferFrom()` [2]. Specifically, when a user intends to reduce the allowed spending amount previously approved from, say, 10 `QIAN` to 1 `QIAN`. The previously approved spender might race to transfer the amount you initially approved (the 10 `QIAN`) and then additionally spend the new amount you just approved (1 `QIAN`). This breaks the user's intention of restricting the spender to the new amount, **not** the sum of old amount and new amount. With the introduction of supporting `metaApprove()`-based meta-transactions, a similar race condition also exists between `metaApprove()`/`transferFrom()`.

In order to properly `approve` tokens, there also exists a known workaround: users can utilize the `increaseApproval` and `decreaseApproval` non-ERC20 functions on the token versus the traditional `approve` function.

```
    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        require(owner != address(0), "QIAN: approve from the zero address");
        require(spender != address(0), "QIAN: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
```

Listing 3.25: QIAN.sol

Recommendation Add the suggested workaround functions `increaseApproval()`/`decreaseApproval()`. However, considering the difficulty and possible lean gains in exploiting the race condition, we also think it is reasonable to leave it as is.

3.16 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-016
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Governance`, `Msign`
- Category: Time and State [14]
- CWE subcategory: CWE-663 [8]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` exploit, and the recent `Uniswap/Lendf.Me` hack.

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `Msign` as an example, the `execute()` function (see the code snippet below) is provided to externally call a contract that is approved by a majority of authorized signers. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 91) starts before effecting the update on internal states (line 94), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `execute()` function.

```
function execute(bytes32 id)
    public
    msignauth(id)
    returns (bool success, bytes memory result)
{
    require(proposals[id].done == 0, "Sign.execute/duplicate execute");
    _weight = 1;
    (success, result) = proposals[id].code.call(proposals[id].data);
    require(success, "Sign.execute/failed");
    _weight = 0;
    proposals[id].done = 1;
    emit Execute(msg.sender, id);
}
```

Listing 3.26: Msign.sol

Another similar violation can be found in the `execute()` routine of the `Proposal` contract.

```
function execute() public auth returns (bool success, bytes memory result) {
    require(exp <= now, "Proposal.execute/vote unexpired");
    require(status == ACTIVATED, "Proposal.execute/unexpected status");
    require(hit == top, "Proposal.execute/unexpected winner");
    (success, result) = action();
    require(success, "Proposal.execute/failed");
    status = EXECUTED;
}
```

Listing 3.27: Proposal.sol

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. The above two functions can be revised as follows:

```
function execute(bytes32 id)
    public
    msignauth(id)
    returns (bool success, bytes memory result)
{
    require(proposals[id].done == 0, "Sign.execute/duplicate execute");
    proposals[id].done = 1;
    _weight = 1;
    (success, result) = proposals[id].code.call(proposals[id].data);
    require(success, "Sign.execute/failed");
    _weight = 0;
    emit Execute(msg.sender, id);
}
```

Listing 3.28: Msign.sol (revised)

```
function execute() public auth returns (bool success, bytes memory result) {
    require(exp <= now, "Proposal.execute/vote unexpired");
```

```
        require(status == ACTIVATED, "Proposal.execute/unexpected status");
        require(hit == top, "Proposal.execute/unexpected winner");
        status = EXECUTED;
        (success, result) = action();
        require(success, "Proposal.execute/failed");
    }
```

Listing 3.29: Proposal.sol (revised)

3.17 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0;` instead of `pragma solidity >=0.6.0;`.

In addition, there is a known compiler issue that in all 0.5.x solidity prior to `Solidity 0.5.17`. Specifically, a private function can be overridden in a derived contract by a private function of the same name and types. Fortunately, there is no overriding issue in this code, but we still recommend using `Solidity 0.5.17` or above.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet.

4 | Conclusion

In this audit, we thoroughly analyzed the QIAN 2.0 design and implementation. The system presents a unique offering in current stablecoin ecosystem and we are impressed by the design and implementation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [20, 21, 22, 23, 25].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [26] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

[3] MITRE. CWE-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.

[4] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[5] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[6] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[7] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

[8] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[9] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

[10] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[11] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[12] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[13] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[14] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[15] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[16] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[17] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[18] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[19] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[20] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[21] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[22] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[23] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[24] PeckShield. PeckShield Inc. https://www.peckshield.com.

[25] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[26] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.