SOFTWARE AUDIT REPORT

for

KCC

Prepared By: Patrick Lou

PeckShield

May 6, 2022

Document Properties

Client	KCC
Title	Software Audit Report
Target	KCC - Genesis
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 6, 2022	Xuxian Jiang	Final Release
1.0-rc1	April 19, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related code of the KuCoin Community Chain (KCC) Genesis contracts as well as the KCC update, we outline in the report our systematic approach to evaluate potential security issues in the implementation, expose possible semantic inconsistencies between the code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About KCC

KuCoin Community Chain (KCC) is a high-performance and decentralized public chain built by the fans of KCC and KuCoin's fan communities. It is designed to have a number of noted characteristics: (1) It is fully compatible with EVM with extremely low migration costs; (2) KuCoin Token (KCS) will serve as the only core fuel and native token for KCC and can be used in scenarios such as gas fee payment; (3) A block-production time of every 3 seconds results in faster transaction confirmation and higher chain performance; and (4) The chain adopts the Proof of Staked Authority (PoSA) consensus algorithm with high efficiency, security and stability. This audit covers the update to the KCC as well as the built-in genesis contracts. The basic information of the audited contracts is shown in Table 1.1.

Table 1.1: Basic Information of the KCC chain

Item	Description
Client	KCC
Website	https://kcc.io
Module	KCC - Genesis
Coding Language	Go/Solidity
Audit Method	Fuzzing, White-box
Latest Audit Report	May 6, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used

in this audit. Note that in this audit, we focus on the following subdirectory `posa` and its interactions with surrounding components, e.g., `genesis-contracts`.

- https://github.com/kcc-community/kcc.git (8b8c947)
- https://github.com/kcc-community/kcc-genesis-contracts.git (1a7a703)

1.2 About PeckShield

PeckShield Inc. [1] 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 security audits. We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

In the first phase of auditing KCC - Genesis, we use fuzzing to find out the corner cases that may not be covered by in-house testing. However, our major efforts are in white-box auditing, in which PeckShield security auditors manually review KCC - Genesis design and source code, analyze them for any potential issues, and follow up with issues found in the fuzzing phase. If necessary, we design and implement individual test cases to further reproduce and verify the issues. In the following subsections, we will introduce the risk model as well as the audit procedure adopted in this report.

Table 1.2: Vulnerability Severity Classification

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

1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [2]:

- 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*, and *Low* shown in Table 1.2.

1.3.2 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique of discovering software vulnerabilities by systematically finding and providing possible inputs to the target program, and then monitoring the program execution for crashes (or any unexpected results). In the first phase of our audit, we use fuzzing to find out possible corner cases or unusual inter-module interactions that may not be covered by in-house testing. As one of the most effective methods for exposing the presence of possible vulnerabilities, fuzzing technology has been the first choice for many security researchers in recent years. At present, there are many fuzzy testing tools and supporting software, which can help security personnels to conduct fuzzing and find vulnerabilities more efficiently. Based on the characteristics of the KCC - Genesis, we use AFL [3] as the primary tool for fuzz testing.

AFL (American Fuzzy Lop) is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. Since its inception, AFL has gained growing popularity in the industry and has proved its effectiveness in discovering quite a few significant software bugs in a wide range of major software projects. The basic process of AFL fuzzing is as follows:

- Generate compile-time instrumentation to record information such as code execution path;
- Construct some input files to join the input queue, and change input files according to different strategies;
- Files that trigger a crash or timeout when executing an input file are logged for subsequent analysis;

- Loop through the above process.

Throughout the AFL testing, we will reproduce each crash based on the crash file generated by AFL. For each reported crash case, we will further analyze the root cause and check whether it is indeed a vulnerability. Once a crash case is confirmed as a vulnerability of the KCC - Genesis, we will further analyze it as part of the white-box audit.

1.3.3 White-box Audit

After fuzzing, we continue the white-box audit by manually analyzing source code. Here we test target software's internal structure, design, coding, and we focus on verifying the flow of input and output through the application as well as examining possible design and implementation trade-offs for strengthened security. PeckShield auditors first fully review and understand the source code, then create specific test cases, execute them and analyze the results. Issues such as internal security loopholes, unexpected output, broken or poorly structured paths, etc., will be inspected under close scrutiny.

Blockchain is a secure method of creating a distributed database of transactions, and three major technologies of blockchain are cryptography, decentralization, and consensus model. Blockchain does come with unique security challenges, and based on our understanding of blockchain general design, we in this audit divide the blockchain software into the following major areas and inspect each area accordingly:

- Data and state storage, which is related to the database and files where blockchain data are saved.
- P2P networking, consensus, and transaction model in the networking layer. Note that the consensus and transaction logic is tightly coupled with networking.
- VM, account model, and incentive model. This is essentially the execution and business layer of the blockchain, and many blockchain business specific logics are implemented here.
- System contracts and services. These are system-level, blockchain-wide operation management contracts and services.
- SDK Security. These are include additional SDK modules and example code for developer community distribution and adoption.
- Others. This includes any software modules that do not belong to above-mentioned areas, such as common crypto or other 3rd-party libraries, best practice or optimization used in other software projects, design and coding consistency, etc.

Table 1.3: The Full List of Audited Items (Part I)

Category	Check Item
Data and State Storage	Blockchain Database Security
	Database State Integrity Check
Node Operation	Default Configuration Security
	Default Configuration Optimization
	Node Upgrade And Rollback Mechanism
Node Communication	External RPC Implementation Logic
	External RPC Function Security
	Node P2P Protocol Implementation Logic
	Node P2P Protocol Security
	Serialization/Deserialization
	Invalid/Malicious Node Management Mechanism
	Communication Encryption/Decryption
	Eclipse Attack Protection
	Fingerprint Attack Protection
Consensus	Consensus Algorithm Scalability
	Consensus Algorithm Implementation Logic
	Consensus Algorithm Security
Transaction Model	Transaction Privacy Security
	Transaction Fee Mechanism Security
	Transaction Congestion Attack Protection
VM	VM Implementation Logic
	VM Implementation Security
	VM Sandbox Escape
	VM Stack/Heap Overflow
	Contract Privilege Control
	Predefined Function Security
Account Model	Status Storage Algorithm Adjustability
	Status Storage Algorithm Security
	Double Spending Protection
Incentive Model	Mining Algorithm Security
	Mining Algorithm ASIC Resistance
	Tokenization Reward Mechanism

Table 1.4: The Full List of Audited Items (Part II)

Category	Check Item
System Contracts And Services	Memory Leak Detection
	Use-After-Free
	Null Pointer Dereference
	Undefined Behaviors
	Deprecated API Usage
	Signature Algorithm Security
	Multisignature Algorithm Security
SDK Security	Using RPC Functions Security
	PrivateKey Algorithm Security
	Communication Security
	Function integrity checking code
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	White Paper And Code Implementation Uniformity

Based on the above classification, we show in Table 1.3 and Table 1.4 the detailed list of the audited items in this report.

To better describe each issue we identified, we also categorize the findings based on Common Weakness Enumeration (CWE-699) [4], which is a community-developed list of software weakness types to better classify and organize weaknesses around concepts frequently encountered in software development. We use the CWE categories in Table 1.5 to classify our findings.

Table 1.5: 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 Logic	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.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given blockchain software. In other words, the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of blockchain software. Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the KuCoin Community Chain (KCC) Genesis contracts as well as the KCC update. As mentioned earlier, we studied in the first phase of our audit the given source code (including related libraries) 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 tools. After that, we manually review business logic, examine system operations, and place operation-specific aspects under scrutiny to uncover possible pitfalls and/or bugs.

Table 2.1: The Severity of Our Findings

Severity	# of Findings
Critical	0
High	1
Medium	2
Low	1
Informational	1
Total	5

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 modules. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined several issues of varying severities that need to be brought up and paid more attention to. These issues are categorized in the above Table 2.1. 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, the audited contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (as shown in Table 2.2), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.2: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Validator Pool Manager Accounting in Validators	Business Logic	Resolved
PVE-002	High	Possible Overwrite of Active Validators in updateActiveValidatorSet()	Business Logic	Resolved
PVE-003	Medium	Incorrect Validator Reward Claims in addValidator()	Coding Practices	Resolved
PVE-004	Informational	Possible Double Initialization From Initializer Reentrancy	Time and State	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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. Please refer to Section 3 for details.

3 Detailed Results

3.1 Proper Validator Pool Manager Accounting in Validators

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Validators`
- Category: Business Logic [5]
- CWE subcategory: CWE-666 [6]

Description

The KCC blockchain is initially based on the `go-ethereum` fork, but with significant enhancements to address various issues. Moreover, the update includes the separation of `manager` and `validator` to mitigate potential risks of exposed `validator` keys for block-producing node operations. While examining the `manager`-related accounting, we notice the current implementation needs to be improved.

To elaborate, we show below the `initialize()` routine in the `Validators` system contract. As the name indicates, this routine configures a number of system-wide parameters, including the initial set of `validator`. With the separation of `manager` and `validator`, we notice the need to properly update the `manager`-related accounting. In other words, for each `manager`, the respective `managedValidatorInfo` state needs to be properly updated with the managed `validator`.

```
157     function initialize(
158         address[] calldata _validators,
159         address[] calldata _managers,
160         uint256[] calldata _feeShares,
161         address _admin,
162         address _validatorsContract,
163         address _punishContract,
164         address _proposalContract,
165         address _reservePool
166     ) external initializer {
167         require(
168             _validators.length == _feeShares.length && _validators.length == _managers.
                length && _validators.length > 0,
```

```
169             "invalidate validator and it's manager"
170         );
171
172         revokeLockingDuration = 3 days;
173         marginLockingDuration = 15 days;
174         feeSetLockingDuration = 1 days;
175         maxPunishmentAmount = 100 ether;
176         minSelfBallots =  10000 ;
177
178         require(address(this).balance >= minSelfBallots.mul(_validators.length).mul(
                VOTE_UNIT), "no enough kcs in validators contract");
179
180         _Admin_Init(_admin);
181         _setAddresses(
182             _validatorsContract,
183             _punishContract,
184             _proposalContract,
185             _reservePool
186         );
187         __ReentrancyGuard_init();
188
189
190         //
191         for (uint256 i = 0; i < _validators.length; ++i) {
192             address val = _validators[i];
193             uint256 feeShares = _feeShares[i];
194
195             // update PoolInfo
196             PoolInfo storage pool = poolInfos[val];
197             pool.manager = _managers[i];
198             pool.validator = val;
199             pool.selfBallots = minSelfBallots;
200             pool.feeShares = feeShares;
201             pool.pendingFee = 0;
202             pool.feeDebt = 0;
203             pool.lastRewardBlock = block.number;
204             // solhint-disable not-rely-on-times
205             pool.feeSettLockingEndTime = block.timestamp.add(
206                 feeSetLockingDuration
207             );
208             pool.suppliedBallots = minSelfBallots;
209             pool.accRewardPerShare = 0;
210             pool.voterNumber = 0;
211             pool.electedNumber = 0;
212             pool.enabled = true;
213
214             // Update Candidate Info
215             Description storage desc = candidateInfos[val];
216             desc.details = "";
217             desc.email = "";
218             desc.website = "";
```

```
220             _sortedEnabledValidators.improveRanking(poolInfos, val);
221             if(activeValidators.length < MAX_VALIDATORS){
222                 activeValidators.push(val);
223             }
224             totalBallot = totalBallot.add(pool.suppliedBallots);
225         }
226     }
```

Listing 3.1: `Validators::initialize()`

Recommendation Revise the above `initialize()` routine to properly update the `manager` accounting.

Status This issue has been fixed in the following commit hash: `1a7a703`.

3.2 Possible Overwrite of Active Validators in updateActiveValidatorSet()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Validators`
- Category: Business Logic [5]
- CWE subcategory: CWE-666 [6]

Description

As mentioned in Section 3.1, the `KCC` blockchain is initially based on the `go-ethereum` fork, but with significant enhancements to address various issues. In particular, the unique adoption of the `Proof of Staked Authority` (`PoSA`) consensus algorithm has a number of pre-deployed system contracts that are instantiated at the first block production. In the following, we examine these system contracts.

To elaborate, we show below a core routine `updateActiveValidatorSet()` in the `Validators` system contract. This routine implements a rather straightforward logic in updating the active validator set. Though this routine is permissioned in only allowing for being invoked from the current miner or the block-producing validator, it blindly trusts the miner without validating the given new set of active validators is consistent with the current top elected validators.

```
527     function updateActiveValidatorSet(address[] calldata newSet, uint256 epoch)
528         external
529         override
530         onlyMiner
531         onlyBlockEpoch(epoch)
532     {
533         operationsDone[block.number][Operation.UpdatedValidators] = true;
```

```
535         require(newSet.length > 0 && newSet.length <= MAX_VALIDATORS, "invalid length of
                newSet array");

537         activeValidators = newSet; // FIXME: gas cost ?
538     }
```

Listing 3.2: Validators :: updateActiveValidatorSet ()

Because of the blind trust, an inconsistent active set may be crafted to vote for/against current proposals. As a result, it can largely determine the results of ongoing proposals. Even worse, since the active validator set influences the current block reward distribution, the routine can also be manipulated to steal block reward (or profit) from legitimate validators.

Recommendation To address the issue at the root cause, it is recommended not to rely on external inputs for the new validator set.

Status This issue has been fixed in the following commit hash: `1a7a703`.

3.3 Incorrect Validator Reward Claims in addValidator()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `KCC`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [8]

Description

By design, each `validator` will be rewarded for each minted block and the system contract `Validators` provides a helper routine `_validatorClaimReward()` to allow to claim the reward for a given `validator`. While examining the `reward`-related logic, we notice the current claim needs to be improved.

To elaborate, we show below this `_validatorClaimReward()` routine. This routine implements a rather straightforward logic in computing the pending reward and transferring it out. It comes to our attention the recipient for the claimed reward is the `msg.sender`, not the intended `validator`! By design, we need to properly send the reward to the `validator` or its `manager`!

```
1064     function _validatorClaimReward(address _val) internal {
1065         PoolInfo storage pool = poolInfos[_val];

1067         //
1068         uint256 pending = _calculateValidatorPendingReward(_val); // roundoff error -
1069         if (pending > 0) {
1070             _safeTransfer(pending);
1071         }
1072         //
```

```
1073            // roundoff error -
1074            pool.selfBallotsRewardsDebt = pool.selfBallots.mul(pool.accRewardPerShare).div(1
                    e12);
1075            emit ValidatorClaimReward(_val, pending);
1076        }
```

Listing 3.3: `Validators::_validatorClaimReward()`

Recommendation Revise the above `_validatorClaimReward()` routine to properly reward the `validator` (or its `manager`).

Status This issue has been fixed in the following commit hash: `1a7a703`.

3.4 Possible Double Initialization From Initializer Reentrancy

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [10]

Description

The KCC system contracts adopt a flexible approach for contract initialization, so that the initialization task does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the `initializer()` modifier that protects an initializer function from being invoked twice. It becomes known that the current version of the used `OpenZepplin` reference has an issue that makes it possible to re-enter `initializer()`-protected functions. In particular, for this to happen, one call may need to be a nested-call of the other, or both calls have to be subcalls of a common `initializer()`-protected function.

The reentrancy can be dangerous as the initialization is not part of the proxy construction, and it becomes possible by executing an external call to an untrusted address. As part of the fix, there is a need to forbid `initializer()`-protected functions to be nested when the contract is already constructed.

To elaborate, we show below the current `initializer()` implementation as well as the fixed implementation.

```
37      modifier initializer() {
38          require(_initializing  _isConstructor()  !_initialized, "Initializable: contract
                is already initialized");
39
40          bool isTopLevelCall = !_initializing;
41          if (isTopLevelCall) {
```

```
42              _initializing = true;
43              _initialized = true;
44          }
45
46          _;
47
48          if (isTopLevelCall) {
49              _initializing = false;
50          }
51      }
```

Listing 3.4: `Initializable::initializer()`

```
37      modifier initializer() {
38          require(_initializing? _isConstructor() : !_initialized, "Initializable:
                contract is already initialized");
39
40          bool isTopLevelCall = !_initializing;
41          if (isTopLevelCall) {
42              _initializing = true;
43              _initialized = true;
44          }
45
46          _;
47
48          if (isTopLevelCall) {
49              _initializing = false;
50          }
51      }
```

Listing 3.5: Revised `Initializable::initializer()`

Recommendation Enforce the `initializer()` modifier to prevent it from being re-entered.

Status The issue has been resolved as the current implementation does not invoke any external contracts.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Validators`
- Category: Security Features [11]
- CWE subcategory: CWE-287 [12]

Description

In the `KCC Genesis` smart contracts, there is a privileged `admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and pool adjustment). It also has the privilege to control or govern the flow of assets managed by the system contracts. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
229     function setMinSelfBallots(uint256 _min) external onlyAdmin {
230         require(_min != minSelfBallots, "Validators: No change detected.");
231
232         minSelfBallots = _min;
233         emit SetMinSelfBallots(_min);
234     }
235
236     function setMaxPunishmentAmount(uint256 _max) external onlyAdmin {
237         require(_max != maxPunishmentAmount, "Validators: No change detected.");
238         maxPunishmentAmount = _max;
239
240         emit SetMaxPunishmentBallots(_max);
241     }
242
243     function setRevokeLockingDuration(uint256 _lockingDuration)
244         external
245         onlyAdmin
246     {
247         require(
248             _lockingDuration != revokeLockingDuration,
249             "Validators: No change detected."
250         );
251
252         revokeLockingDuration = _lockingDuration;
253         emit SetRevokeLockingDuration(_lockingDuration);
254     }
255
256     function setFeeSetLockingDuration(uint256 _lockingDuration)
257         external
258         onlyAdmin
259     {
260         require(
```

```
261             _lockingDuration != feeSetLockingDuration,
262             "Validators: No change detected."
263         );
264
265         feeSetLockingDuration = _lockingDuration;
266         emit SetFeeSetLockingDuration(_lockingDuration);
267     }
```

Listing 3.6: Example `Setters` in the `Validators` Contract

Notice that the privilege assignment is necessary and consistent with the contract design. In the meantime, the extra power to the owner may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these `onlyAdmin` privileges explicit or raising necessary awareness among contract users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 Conclusion

In this security audit, we have analyzed the updates to the KCC blockchain, which is a high-performance and decentralized public chain built by the fans of KCC and KuCoin's fan communities. During the first phase of our audit, we study the source code and run our in-house static analysis tools through the codebase. A list of potential issues were found and we have accordingly developed various test cases to reproduce and verify each of them. After further analysis and internal discussion, we determine that a number of issues need to be brought up and paid more attention to, which are reported in Sections 2 and 3.

Our impression through this audit journey is that the codebase is developed on top of the solid go-ethereum with the unique contribution of a POSA consensus algorithm. In the meantime, the identified issues are promptly confirmed and fixed. We'd like to commend the team for quickly addressing issues found during the audit process. Also, as expressed in Section 1.4, we appreciate any constructive feedback or suggestions about this report.

References

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

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

[3] Lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.

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

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

[6] MITRE. CWE-666: Operation on Resource in Wrong Phase of Lifetime. https://cwe.mitre.org/data/definitions/666.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[10] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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