



SOFTWARE AUDIT REPORT

for

CONFLUX



Prepared By: Shuxiao Wang

Hangzhou, China

Nov. 4, 2020

Document Properties

Client	Conflux
Title	Software Audit Report
Target	Conflux Blockchain
Version	1.0
Author	Edward Lo
Auditors	Edward Lo, Ruiyi Zhang, Xudong Shao
Reviewed by	Jeff Liu, Chiachih Wu, Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	Nov. 4, 2020	Edward Lo	Final Release

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	4
1.1	About Conflux Blockchain	4
1.2	About PeckShield	5
1.3	Methodology	5
1.3.1	Risk Model	5
1.3.2	Fuzzing	6
1.3.3	White-box Audit	7
1.4	Disclaimer	10
2	Findings	11
2.1	Finding Summary	11
2.2	Key Findings	12
3	Detailed Results	13
3.1	Specification and Implementation Discrepancy #1 - BEGINSUB	13
3.2	Missed Sanity Checks in set_admin()	14
3.3	Specification and Implementation Discrepancy #2 - destroy()	15
3.4	Specification and Implementation Discrepancy #3 - set_sponsor_for_gas()	17
3.5	Business Logic Error in GetStakingBalance()	18
3.6	Specification and Implementation Discrepancy #4 - process_rewards_and_fees()	20
4	Conclusion	22
	References	23

1 | Introduction

Given the opportunity to review the **Conflux Blockchain** design document and related source code, we outline in this report our systematic method to evaluate potential security issues in the Conflux Blockchain implementation, expose possible semantic inconsistencies between the source code and the design specification, and provide additional suggestions and recommendations for improvement. Our results show that the given branch of Conflux Blockchain can be further improved due to the presence of several issues related to either security or performance. This document describes our audit results in detail.

1.1 About Conflux Blockchain

Conflux Blockchain is a public blockchain system designed by Conflux Labs. Conflux Network is a scalable and decentralized blockchain system with high throughput and fast confirmation. It operates a novel consensus protocol called GHOST which optimistically processes concurrent blocks without discarding any as forks, and adaptively assigns heterogeneous weights to blocks based on their topologies in the ledger structure (called Tree-Graph).

The basic information of Conflux Blockchain is shown in Table 1.1, and its Git repository and the commit hash value (of the audited branch) are in Table 1.2.

Table 1.1: Basic Information of Conflux Blockchain

Item	Description
Issuer	Conflux
Website	https://confluxnetwork.org/
Type	Conflux Blockchain
Platform	Rust
Audit Method	White-box
Latest Audit Report	Nov. 4, 2020

Table 1.2: The Commit Hash List Of Audited Branches

Git Repository	Commit Hash Of Audited Branch
github.com/Conflux-Chain/conflux-rust	49998069c82208306584b4f841e9f705258e0d3c

1.2 About PeckShield

PeckShield Inc. [1] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the 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 Conflux Blockchain, we use fuzzing to find out the corner cases that may not be covered by in-house testing. Next we do white-box auditing, in which PeckShield security auditors manually review Conflux Blockchain 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 1.3: Vulnerability Severity Classification

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

1.3.1 Risk Model

To standardize the evaluation, we define the following terminology based on the 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 1.3.

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 Conflux Blockchain, 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 Conflux Blockchain, we will further analyze it as part of the white-box audit.

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

Category	Check Item
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
	Tokenization Reward Mechanism

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:

- Consensus, which is GHAST protocol based on Tree-Graph.

Table 1.5: 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
Others	Third Party Library Security
	Memory Leak Detection
	Exception Handling
	Log Security
	Coding Suggestion And Optimization
	White Paper And Code Implementation Uniformity

- 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.
- 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.

Based on the above classification, we show in Table 1.4 and Table 1.5 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 1.6 to classify our findings.

Table 1.6: 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.
Input Validation Issues	Weaknesses in this category are related to a software system's input validation components. Frequently these deal with sanitizing, neutralizing and validating any externally provided inputs to minimize malformed data from entering the system and preventing code injection in the input data.
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 audit does not give any warranties on finding all possible security issues of the given blockchain software, i.e., 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 Finding Summary

Here is a summary of our findings after analyzing Conflux Blockchain. As mentioned earlier, we in the first phase of our audit studied Conflux Blockchainsource code and ran our in-house static code analyzer through the codebase, and 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 logics, examine system operations, and place operation-specific aspects under scrutiny to uncover possible pitfalls and/or bugs.

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 more than a dozen of issues of varying severities that need to be brought up, which are categorized in Table 2.1. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

Table 2.1: The Severity of Our Findings

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	0	
Informational	5	
Total	6	

2.2 Key Findings

After analyzing all of the potential issues found during the audit, we determined that a number of them need to be brought up and paid more attention to, as shown in Table 2.2. Please refer to Section 3 for detailed discussion of each issue.

Table 2.2: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Info.	Specification and Implementation Discrepancy #1 - BEGINSUB	Coding Practices	Fixed
PVE-002	Medium	Missed Sanity Checks in set_admin()	Business Logic	Fixed
PVE-003	Info.	Specification and Implementation Discrepancy #2 - destroy()	Coding Practices	Confirmed
PVE-004	Info.	Specification and Implementation Discrepancy #3 - set_sponsor_for_gas()	Coding Practices	Confirmed
PVE-005	Info.	Business Logic Error in GetStakingBalance()	Error Conditions, Return Values, Status	Fixed
PVE-006	Info.	Specification and Implementation Discrepancy #4 - process_rewards_and_fees()	Coding Practices	Confirmed

3 | Detailed Results

3.1 Specification and Implementation Discrepancy #1 - BEGINSUB

- ID: PVE-001
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: `core/src/evm/interpreter/mod.rs`
- Category: Coding Practices [5]
- CWE subcategory: CWE-684 [6]

Description

Similar to Ethereum, Conflux operates with an account-based model where every normal account is associated with a balance and each contract account contains the corresponding byte codes as well as the internal states. Moreover, Conflux supports a modified version of Solidity and Ethereum Virtual Machine (EVM) for its smart contracts, so that Ethereum smart contracts could be readily migrated to Conflux. Note that Ethereum introduces `BEGINSUB`, `JUMPSUB`, `RETURNSUB` instructions in EIP-2315, and Conflux implements these instructions with identical behaviors as Ethereum. However, while reviewing the implementation, we notice that the execution of `BEGINSUB` vm opcode, which should mark the entry point to a subroutine, may result in an OOG (Out Of Gas) exception.

```

716 instructions::BEGINSUB => {
717     // BEGINSUB should not be executed. If so, returns
718     // InvalidSubEntry (EIP-2315).
719     return Err(vm::Error::InvalidSubEntry);
720 }
```

Listing 3.1: `core/src/evm/interpreter/mod.rs`

As shown in the code snippet above, the current implementation returns a `InvalidSubEntry` error, leading to a discrepancy between the specification and the implementation.

Recommendation Update the corresponding description in the specification.

Status The issue has been fixed by this commit: `d9571bf`.

3.2 Missed Sanity Checks in set_admin()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `core/src/state/account_entry.rs`
- Category: Business Logic [7]
- CWE subcategory: CWE-708 [8]

Description

Conflux provides three built-in contracts for system maintenance and on-chain governance. Specifically, Conflux implements a sponsorship mechanism to subsidize the usage of smart contracts and an `AdminControl` mechanism for better maintenance of other smart contracts. Also, Conflux introduces the staking mechanism to provide a better way to charge the occupation of storage space (comparing to “pay once, occupy forever”) and help in defining the voting power in decentralized governance. We also note that Conflux provides a `solidity`-like interface for developers.

In this section, we mainly focus on the mechanism of the `AdminControl` contract which records the administrator of every user-established smart contract and handles the destruction on request of corresponding administrators. In particular, the `set_admin` interface of `AdminControl` contract sets the administrator of contract to a given `admin`.

```

271 pub fn set_admin(&mut self, requester: &Address, admin: &Address) {
272     if self.is_contract() && self.admin == *requester {
273         self.admin = admin.clone();
274     }
275 }
```

Listing 3.2: `core/src/state/account_entry.rs`

However, as the specification indicates, the new `admin` should be a normal account. It means contract accounts should not be allowed to be the administrator of other contracts. But the current implementation of `set_admin` does not validate whether the new `admin` account is a contract address or not. As a result, the lack of validation might lead to the loss of `admin` control.

Recommendation Make sure the new `admin` account is a normal account, as shown in the following example code.

```

271 pub fn set_admin(&mut self, requester: &Address, admin: &Address) {
272     if self.is_contract()
273         && self.admin == *requester
274         && (admin.is_user_account_address() || admin.is_null_address())
275     {
276         self.admin = admin.clone();
277     }
```

278 }

Listing 3.3: `core/src/state/account_entry.rs`**Status** The issue has been fixed by this commit: `6da68ea`.

3.3 Specification and Implementation Discrepancy #2 - `destroy()`

- ID: PVE-003
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: `core/src/executive/internal_contract/impls/admin.rs`
- Category: Coding Practices [5]
- CWE subcategory: CWE-684 [6]

Description

As described in Section 3.2, the `AdminControl` contract is introduced for better maintenance of the other smart contracts. According to the specification, the default administrator of a smart contract `c` is the creator of `c`, i.e., the sender `A` of the transaction that causes the creation of `c`. At any time, the administrator `A` of an existing contract `c` has the right to destruct `c` by calling `AdminControl`. But the destruction request should be rejected if the collateral for storage of contract `c` is not zero. However, the current implementation, `destroy()`, is not consistent with the specification.

```

65 pub fn destroy(
66     contract_address: Address, params: &ActionParams, state: &mut State,
67     spec: &Spec, substate: &mut Substate,
68 ) -> vm::Result<()>
69 {
70     debug!("contract_address={:?}", contract_address);
71
72     let requester = &params.original_sender;
73     let admin = state.admin(&contract_address)?;
74     if admin == *requester {
75         suicide(&contract_address, &admin, state, spec, substate)
76     } else {
77         Ok(())
78     }
79 }
```

Listing 3.4: `core/src/executive/internal_contract/impls/admin.rs`

To elaborate, we show above the code snippet of the `destroy()` routine. Note that after the validation of the `requester` (line 74), this routine invokes a helper function named `suicide()` (line 75) to accordingly handle token refunding or burning when a contract address is being destroyed.

```

18 pub fn suicide(
19     contract_address: &Address, refund_address: &Address, state: &mut State,
20     spec: &Spec, substate: &mut Substate,
21 ) -> vm::Result<()>
22 {
23     substate.suicides.insert(contract_address.clone());
24     let balance = state.balance(contract_address)?;
25
26     if refund_address == contract_address || !refund_address.is_valid_address()
27     {
28         // When destroying, the balance will be burnt.
29         state.sub_balance(
30             contract_address,
31             &balance,
32             &mut substate.to_cleanup_mode(spec),
33         )?;
34         state.subtract_total_issued(balance);
35     } else {
36         trace!(target: "context", "Destroying {} -> {} (xfer: {})", contract_address,
37             refund_address, balance);
38         state.transfer_balance(
39             contract_address,
40             refund_address,
41             &balance,
42             substate.to_cleanup_mode(spec),
43         )?;
44     }
45     Ok(())
46 }

```

Listing 3.5: core/src/executive/internal_contract/impls/admin.rs

However, inside `suitide()`, the check for collateral `for storage` is missed, which leads to another discrepancy between the specification and implementation.

Recommendation Either update the corresponding description in the specification or revise the above implementation for better consistency.

Status This issue has been confirmed.

3.4 Specification and Implementation Discrepancy #3 - set_sponsor_for_gas()

- ID: PVE-004
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: `core/src/executive/internal_contract/impls/admin.rs`
- Category: Coding Practices [5]
- CWE subcategory: CWE-684 [6]

Description

As described in Section 3.2, Conflux implements a sponsorship mechanism to subsidize the usage of smart contracts. Thus, a new account with zero balance is able to call smart contracts as long as the execution is sponsored. To achieve that, the built-in `SponsorControl` contract is introduced to record the sponsorship information of smart contracts. In particular, the `SponsorControl` contract keeps the `SponsorInfo` information for each user-established as follows:

- **sponsor for gas**: this is the account that provides the subsidy for gas consumption;
- **sponsor for collateral**: this is the account that provides the subsidy for the collateral for storage;
- **sponsor balance for gas**: this is the balance of subsidy available for gas consumption;
- **sponsor balance for collateral**: this is the balance of subsidy available for the collateral for storage;
- **sponsor limit for gas fee**: this is the upper bound for the gas fee subsidy paid for every sponsored transaction.

Here, both sponsorship for gas and sponsorship for collateral can be updated by calling the `SponsorControl` contract. The code snippet below is responsible for setting and updating the sponsorship for gas. It checks whether the target is an existing contract (lines 19–27). Moreover, the transferred fund should not be less than 1000 times of the new limit, so that it is sufficient to subsidize at least 1000 transactions (lines 31–35).

```

12 pub fn set_sponsor_for_gas(
13     contract_address: Address, upper_bound: U256, params: &ActionParams,
14     spec: &Spec, state: &mut State, substate: &mut Substate,
15 ) -> vm::Result<()>
16 {
17     let sponsor = &params.sender;
18 
```

```

19     if !state.exists(&contract_address)? {
20         return Err(vm::Error::InternalContract("contract address not exist"));
21     }
22
23     if !contract_address.is_contract_address() {
24         return Err(vm::Error::InternalContract(
25             "not allowed to sponsor non-contract account",
26         ));
27     }
28
29     let sponsor_balance = state.balance(&params.address)?;
30
31     if sponsor_balance / U256::from(1000) < upper_bound {
32         return Err(vm::Error::InternalContract(
33             "sponsor should at least sponsor upper_bound * 1000",
34         ));
35     }

```

Listing 3.6: core/src/executive/internal_contract/impls/sponsor.rs

However, as mentioned in the specification, current sponsors should be able to call this contract to transfer funds to increase the sponsor balances directly, or to increase the sponsor limit for gas without transferring new funds. As a result, this is yet another discrepancy between the spec and the implementation.

Recommendation Either update the corresponding description in the specification or revise the above implementation for better consistency.

Status This issue has been confirmed.

3.5 Business Logic Error in GetStakingBalance()

- ID: PVE-005
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: core/src/executive/internal_contract/contracts/staking.rs
- Category: Error Conditions, Return Values, Status Codes [9]
- CWE subcategory: CWE-394 [10]

Description

As described in Section 3.2, there is a built-in staking contract that records the staking information of all accounts. By sending a transaction to this contract, users, including both external users and smart contracts, can deposit and withdraw funds. The deposited funds are also called stakes in the contract. The interest of staked funds is paid at withdrawal time depending on the amount and

staking period of the fund being withdrawn. In Conflux, the staking contract keeps track of staked funds and freezing rules. It also provides a getter interface, i.e., `GetStakingBalance()` to query the staking balance of a given address.

```

124 impl ExecutionTrait for GetStakingBalance {
125     fn execute_inner(
126         &self, input: Address, _: &ActionParams, _spec: &Spec,
127         state: &mut State, _substate: &mut Substate,
128     ) -> vm::Result<U256>
129     {
130         Ok(state.collateral_for_storage(&input)?)
131     }
132 }

```

Listing 3.7: `core/src/executive/internal_contract/contract/staking.rs`

However, this function returns the wrong value (line 130) of `collateral_for_storage`, not the intended `staking_balance`.

Recommendation Return `staking_balance` in `GetStakingBalance(address)`, as shown in the following code example.

```

124 impl ExecutionTrait for GetStakingBalance {
125     fn execute_inner(
126         &self, input: Address, _: &ActionParams, _spec: &Spec,
127         state: &mut State, _substate: &mut Substate,
128     ) -> vm::Result<U256>
129     {
130         Ok(state.staking_balance(&input)?)
131     }
132 }

```

Listing 3.8: `core/src/executive/internal_contract/contract/staking.rs`

Status This issue has been addressed in this commit: [a04ce02](#).

3.6 Specification and Implementation Discrepancy #4 - process_rewards_and_fees()

- ID: PVE-006
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: core/src/consensus/consensus_inner/consensus_executor.rs
- Category: Coding Practices [5]
- CWE subcategory: CWE-684 [6]

Description

Conflux miners are rewarded from two sources: block rewards and transaction fees. In particular, the amount of coins rewarded to miners in every block is set in accordance with a global parameter, base block award, which follows the mining schedule. Based on the base block award, Conflux defines the actual block rewards issued to the author of block B with adjustments. Specifically, Conflux introduces the base factor $BF(B)$ to indicate whether the author of B is eligible to receive any reward. As mentioned in the specification, the quality of block (line 1324) should satisfy the requirement $QUALITY(B) \geq 250 \cdot d_{EPOCH_k}$.

```

1308 // Base reward and anticone penalties.
1309 for (enum_idx, block) in epoch_blocks.iter().enumerate() {
1310     let no_reward = reward_info.epoch_block_no_reward[enum_idx];
1311
1312     if no_reward {
1313         epoch_block_total_rewards.push(U256::from(0));
1314         if debug_record.is_some() {
1315             let debug_out = debug_record.as_mut().unwrap();
1316             debug_out.no_reward_blocks.push(block.hash());
1317         }
1318     } else {
1319         let pow_quality =
1320             VerificationConfig::get_or_compute_header_pow_quality(
1321                 &self.data_man.pow,
1322                 &block.block_header,
1323             );
1324         let mut reward = if pow_quality >= *epoch_difficulty {
1325             base_reward_per_block
1326         } else {
1327             debug!(
1328                 "Block {} pow_quality {} is less than epoch_difficulty {}",
1329                 block.hash(), pow_quality, epoch_difficulty
1330             );
1331             0.into()
1332         };
1333     }

```

Listing 3.9: core/src/consensus/consensus_inner/consensus_executor.rs

However, the requirement in the current implementation is $\text{QUALITY}(B) \geq \mathbf{d}_{\text{EPOCH}_k}$. This leads to a discrepancy between the specification and implementation.

Recommendation Either update the corresponding description in the specification or revise the above implementation for better consistency.

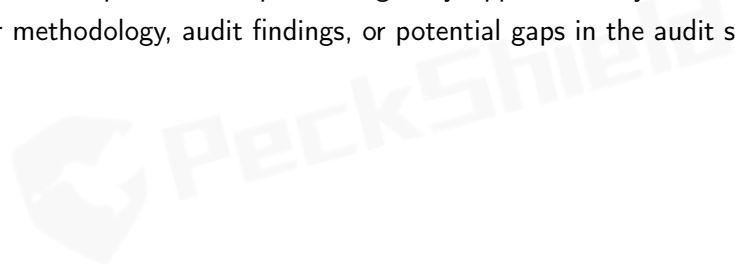
Status This issue has been confirmed.



4 | Conclusion

In this security audit, we have analyzed the source code and system design of the Conflux Blockchain. During the first phase of our audit, we studied the source code and ran our in-house analyzing tools through the codebase, including areas such as virtual machine, consensus algorithm, and transaction model, etc. A list of potential issues were found, and some of them involve unusual interactions among multiple modules, therefore we developed test cases to reproduce and verify each of them. After further analysis and internal discussion, we determined that six issues need to be brought up and paid more attention to, which are reported in Sections [2](#) and [3](#).

Our impression through this audit is that the Conflux Blockchain software is neatly organized and elegantly implemented and those identified issues are promptly confirmed and fixed. We'd like to commend Conflux for a well-done software project, and for quickly fixing issues found during the audit process. Also, to improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in the audit scope/coverage.



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: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE-684: Incorrect Provision of Specified Functionality. <https://cwe.mitre.org/data/definitions/684.html>.
- [7] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [8] MITRE. CWE-708: Incorrect Ownership Assignment. <https://cwe.mitre.org/data/definitions/708.html>.
- [9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.

- [10] MITRE. CWE CATEGORY: Unexpected Status Code or Return Value. <https://cwe.mitre.org/data/definitions/394.html>.

