SMART CONTRACT AUDIT REPORT

for

DEFIBOX

Prepared By: Shuxiao Wang

PeckShield

May 26, 2021

Document Properties

Client	Defibox
Title	Smart Contract Audit Report
Target	Defibox USN
Version	1.0-rc2
Author	Shawn Li
Auditors	Shawn Li, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc2	May 26, 2021	Shawn Li	Release Candidate #2
1.0-rc	May 21, 2021	Shawn Li	Release Candidate

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 design document and related smart contract source code of the **Defibox USN**, we outline in the report 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 contracts 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 Defibox USN

Defibox USN is a decentralized staking stable coin project based on EOS, which supports users to stake EOS, BTC, ETH to generate stable coin USN (The goal is to anchor the dollar price). Through the risk control mechanism of excess staking and liquidation, the system avoids market fluctuations and ensures that there are sufficient staking items to provide value support for each USN. Decentralized operations are based on the smart contract, and data can be checked in real-time on the chain.

The basic information of the Defibox USN is as follows:

Table 1.1: Basic Information of Defibox USN

Item	Description
Issuer	Defibox
Website	https://defibox.io/
Type	EOS Smart Contract
Platform	C++
Audit Method	Whitebox
Latest Audit Report	May 26, 2021

In the following, we show the reviewed files hash and the smart contract code hash(compiled with EOSIO.CDT v1.7.0) used in this audit:

- Code Information v4 (5/26/2021):
  MD5: 0x5c4d4978ea4d4078df990d3d07fe573c
  SHA256: 0x9f6e9c9790d300be0cc8643d884067e637647d7d07da5e241da8ea3531be8803
  Code Hash: 0xac4b81cc1e260f1ffbc038c3e3770c75fc669fb70b62a1a0262ae9d6aab5839b

- Code Information v3 (5/17/2021):
  MD5: 0x61f1bbb38a3bc339edde44b90c94bd40
  SHA256: 0x8e4ea447e4f55115ac1c2d0aa3761a177a0feb00ad1c37d1b8eb06aa9f1a9567
  Code Hash: 0x358d0a94aa5210daf77f76991375b18d8da253d0ece77ceb722a7ca70a977a8f

- Code Information v2 (5/8/2021):
  MD5: 0xb33c115454cb88cd5e8bbb7144f551bf
  SHA256: 0xebdecbcfa106d04a68bed2e30e478cd0da51b425b7ba1e0473c595da18f98cfe
  Code Hash: 0xb716d834016b8ebd08fbdd3dbfecfe0746b7bcdadc00039b3ec0531fc2ad0ee2

- Code Information v1 (4/28/2021):
  MD5: 0x9e7c3a0c1d1aede15f08a0da0becb36a
  SHA256: 0x7e8a4793064174194d1504fa628eaedd663a4dc5f120fef88cea5b3268875a8c
  Code Hash: 0xa9af55b966f69f9b0b95917f687a26f73605f60b15afdc5a020668feb4daf470

1.2 About PeckShield

PeckShield Inc. [7] 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 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).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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.

Table 1.2: Vulnerability Severity Classification

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

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.

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) [5], which is a community-developed list of software weakness types to

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Vulnerability Detection	Uncontrolled Apply Call
	Missed Permission Check
	Fake Token
	Overflows & Underflows
	Fake Transfer Notice
	Transaction Rollback
	Transaction Congestion
	soft_fail State
	hard_fail State
	Abnormal Memo
	Abnormal Resource Consumption
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
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Using Fixed Compiler Version
	Following Other Best Practices

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 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 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 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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Defibox USN implementation. During the first phase of our audit, we study the smart contract source code and run 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	0	
Medium	2	■ ■
Low	1	■
Informational	1	■
Total	4	

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 that 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 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings of The Defibox USN Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Missed Some Actions Dispatch in apply()	Behavioral Issues	Fixed
PVE-002	Informational	Inappropriate Logs in Generate Action	Behavioral Issues	Fixed
PVE-003	Medium	Inappropriate Interest Receiving Account in repay2()	Business Logic	Fixed
PVE-004	Medium	Missed Collateral ID Check in Bid Action	Behavioral Issues	Fixed

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.

3 | Detailed Results

3.1 Missed Some Actions Dispatch in apply()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: usn.cpp
- Category: Behavioral Issues [3]
- CWE subcategory: CWE-440 [1]

Description

Defibox USN is a decentralized staking stable coin contract based on EOS. The `apply()` function on EOS is the entry point of a contract. By modifying the `apply()` function, it can control the processing logic when the contract is called. However, our analysis shows that the `apply()` function of the contract lacks some actions dispatch, so these actions will not be executed.

```
    ACTION bidresult(name user, uint64_t collateral_id, name contract, uint64_t aid,
        uint64_t bid, asset bidfund, asset bidrefund, asset bideos, uint64_t price,
        uint64_t disprice, double discount, bool status, asset remainpledge, asset
        remainissue, time_point_sec start);

    ACTION ratechange(name user, uint64_t collateral_id, time_point_sec start,
        time_point_sec lastupdate, uint64_t loan, uint64_t interest, uint64_t rate);

    ACTION buyrexlog(name user, asset quantity, asset rex_value);

    ACTION sellrexlog(name user, asset quantity, asset rex_value);
```

Listing 3.1: usn.hpp

```
extern "C" {
    void apply(uint64_t receiver, uint64_t code, uint64_t action) {
        auto self = receiver;

        if (code == self) {
            switch (action) {
```

```
                EOSIO_DISPATCH_HELPER(usn, (init)(adjust)(sellnext)(bidnext)(clear)(
                    clearresult)
                (repayresult)(bidresult)(ratechange)(buyrex)(sellrex)(sellallrex)(
                    setstate)(calinterest)
                (createlog)(adjustlog)(withdrawlog)(generatelog)(adddepositlog)(repaylog
                    )(incomelog)(checkbalance)(proxyto)(addreward)(delreward)(
                    sendrewards)
                (syncaccounts)(createtoken)(removetoken)(modifytoken)(setinterest))
                /*(fix)(reset)(withdraw) */
            }
        } else {
            if (action == name("transfer").value) {
                usn ptop(name(receiver), name(code), datastream<const char *>(nullptr,
                    0));
                const auto t = unpack_action_data<transfer_args>();
                ptop.handle_transfer(t.from, t.to, t.quantity, t.memo, name(code));
            }
        }
    }
}
```

Listing 3.2: usn::apply()

Specifically, we show above the related `apply()` function. This function is designed and implemented to dispatch all actions. However, the `buyrexlog` and `sellrexlog` are missed. They are declared on lines 133 – 135 in the `usn.hpp` file. In this way, when the relevant action is called, the function in the contract will not be actually executed.

Recommendation It is recommended to add the `buyrexlog` and `sellrexlog` declarations to the apply() function.

Status The issue has been fixed in v3 code (`61f1bbb3`).

3.2 Inappropriate Logs in Generate Action

- ID: PVE-002
- Severity: Informational
- Likelihood: Low
- Impact: N/A
- Target: `usn.cpp`
- Category: Behavioral Issues [3]
- CWE subcategory: CWE-440 [1]

Description

The Defibox USN contract supports users to stake EOS, BTC, ETH to generate stable coin USN. Many actions are used in the contract to record logs. For example, `incomelog` is generated when the contract receives tokens. However, our analysis shows that even if the user fails to generate USN and the contract returns the user's token, an `incomelog` will still be generated.

To elaborate, we show part of the implementation code of the `generate()` function below:

```
555     if (status) {
556         result_str = string("USN issue success");
557         action(
558             permission_level{USNTOKEN_CONTRACT, "active"_n},
559             USNTOKEN_CONTRACT,
560             name("issue"),
561             make_tuple(from, usn_quantity, string("USN issue"))
562         ).send();
563
564         action(
565             permission_level{_self, "active"_n},
566             _self,
567             name("generatelog"),
568             make_tuple(from, collateral_id, contract, rate, quantity, usn_quantity,
                    result_str, status, debt_itr->pledge, debt_itr->issue, price, now)
569         ).send();
570
571     } else {
572         result_str = string("generate USN failure, refund deposit");
573         inline_transfer(contract, _self, from, quantity, result_str);
574         sub_collateral_balance(contract, quantity);
575     }
576
577     if (writelog) {
578         action(permission_level{_self, "active"_n}, _self, name("incomelog"), make_tuple
                (from, collateral_id, contract, quantity)).send();
579     }
```

Listing 3.3: usn::generate()

From the above code, we notice that the `generate()` function receives collateral and issues USN. If the `status` is true (line 555), it means that the generation is successful. If it is false, the contract will return the collateral tokens. However, `incomelog` will always be generated (lines 577 – 579).

Recommendation It is recommended to call `incomelog` action after actually receiving the tokens.

Status The issue has been fixed in v3 code (`61f1bbb3`).

3.3 Inappropriate Interest Receiving Account in repay2()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `usn.cpp`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

Users deposit tokens as collateral to generate USN and need to pay interest. The Defibox USN contract supports users to repay interest through collateral. However, we notice that the account that receives the interest is still the user's account. As a result, the users do not pay interest.

```
void usn::repay2(name from, uint64_t collateral_id, asset quantity, uint64_t rate) {
    static const time_point_sec now{current_time_point().sec_since_epoch()};

    auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported collateral");

    if (rate != 0) {
        check(rate >= collateral_itr->min_rate, string("Invalid memo, rate should >= ") + to_string(collateral_itr->min_rate));
    }

    uint64_t price = get_price(collateral_itr);
    check_price(collateral_itr, price);

    string result_str;

    asset total_usn_quantity = asset(0, USN_SYMBOL);
    asset usn_of_interest = asset(0, USN_SYMBOL);
    asset pledge_of_interest = asset(0, collateral_itr->sym);
    asset total_pledge_quantity = asset(0, collateral_itr->sym);

    debts debt_tbl(_self, collateral_itr->id);
    auto debt_itr = debt_tbl.require_find(from.value, "not found cdp");

    debt_tbl.modify(debt_itr, same_payer, [&](auto &cdp) {
        auto repay_amount = quantity.amount;

        uint64_t total_interest = 0;
        uint64_t total_capital = 0;


        while (!cdp.records.empty() && repay_amount > 0) {
            uint64_t total_unpay_amount = cdp.records.front().second.capital;


```

```
            uint64_t this_pay;
            if (repay_amount >= total_unpay_amount) {
                this_pay = total_unpay_amount;
                repay_amount -= this_pay;
            } else {
                this_pay = repay_amount;
                repay_amount = 0;
            }

            double ratio = (double(this_pay) / cdp.records.front().second.capital);

            uint64_t pay_capital = this_pay;

            uint64_t pay_interest = get_interest(this_pay, cdp.records.front().first,
                cdp.records.front().second.last_update, collateral_itr->interest);

            uint64_t unpay_interest = cdp.records.front().second.unpay_interest;
            uint64_t pay_unpay_interest = 0;
            if (unpay_interest > 0) {
                pay_unpay_interest = (uint64_t)ceil((unpay_interest)*ratio);
                pay_interest += pay_unpay_interest;
            }


            total_interest += pay_interest;
            total_capital += pay_capital;

            check(pay_capital <= cdp.records.front().second.capital, "repay error");

            action(
                permission_level{_self, "active"_n},
                _self,
                name("repayresult"),
                make_tuple(cdp.user, collateral_id, cdp.records.front().second.capital,
                    pay_capital, pay_interest, cdp.records.front().first)
            ).send();

            if (cdp.records.front().second.capital == pay_capital) {
                cdp.records.pop_front();
            } else if (cdp.records.front().second.capital > pay_capital) {
                cdp.records.front().second.capital -= pay_capital;

                if (pay_unpay_interest < cdp.records.front().second.unpay_interest) {
                    cdp.records.front().second.unpay_interest -= pay_unpay_interest;
                } else {
                    cdp.records.front().second.unpay_interest = 0;
                }
            }
        }

        if (total_interest > 0) {

```

```
895             pledge_of_interest.amount = calc_pledge_amount2(asset(total_interest,
                    USN_SYMBOL), 1e4, price, pledge_of_interest.symbol);
896
897             print_f("pledge of interest: %, total interest: %, price: %\n",
                    pledge_of_interest, total_interest, price);
898
899             check(pledge_of_interest.amount > 0, "pledge of interest == 0");
900             transfer_to(collateral_itr, from, pledge_of_interest, string("USN interest
                    fee"));
901         }
902
903         if (repay_amount > 0) {
904             asset repay_refund_quantity = asset(repay_amount, USN_SYMBOL);
905             inline_transfer(USNTOKEN_CONTRACT, _self, cdp.user, repay_refund_quantity,
                    string("repay refund"));
906         }
```

Listing 3.4: usn::repay2()

From the above code, we notice that the `usn::repay2()` function calculates the user's accrued interest and transfers it to the specified account. However, the account that receives the interest is not the `USNFEE_ACCOUNT` defined in the contract, but the user's account (line 900). This results in the user not paying interest.

Recommendation It is recommended to transfer the interest to the `USNFEE_ACCOUNT` account.

Status The issue has been fixed in v3 code (`61f1bbb3`).

3.4 Missed Collateral ID Check in Bid Action

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: usn.cpp
- Category: Behavioral Issues [3]
- CWE subcategory: CWE-440 [1]

Description

The Defibox USN contract allows users to generate USN by leveraging collateral assets. The system avoids market fluctuations and ensures that there are sufficient staking items to provide value support for each USN. When the token price drops and the user's collateral is insufficient, the collateral of the account may be liquidated. Users can bid on these auctioned assets to obtain revenue. Our analysis shows that when bidding, the contract does not check whether the collateral id declared by the bidding user is the same as the collateral id being auctioned.

Specifically, we revisit the `usn::bid()` function that implements the functionality of bidding for liquidated assets. As shown in the following code, users need to provide `collateral_id` (line 986) and auction id (`aid`) when bidding. According to the `aid`, the auctioned collateral assets can be queried from the auctions table. But the function does not check whether the `collateral_id` and the auctioned collateral id are the same.

```
986  void usn::bid(name from, uint64_t collateral_id, asset quantity, uint64_t aid) {
987      auto auction_itr = _auctions.require_find(aid, "no auction found");
988
989      static const time_point_sec now{current_time_point().sec_since_epoch()};
990
991      auto collateral_itr = _collaterals.require_find(collateral_id, "Unsupported
             collateral");
992      uint64_t price = get_price(collateral_itr, false);
993      check_price(collateral_itr, price);
994
995      asset bid_fund = asset(0, USN_SYMBOL);
996      asset bid_refund = asset(0, USN_SYMBOL);
997
998
999      if (auction_itr->remain_issue >= quantity) {
1000         bid_fund = quantity;
1001     } else {
1002         bid_fund = auction_itr->remain_issue;
1003         bid_refund = quantity - bid_fund;
1004     }
1005
1006     auto time_sec_diff = now.sec_since_epoch() - auction_itr->create_time.
             sec_since_epoch();
1007     double discount_rate = get_discount(time_sec_diff);
```

```
    uint64_t discount_price = price * discount_rate;

    uint64_t pledge_amount = calc_pledge_amount(bid_fund, 1e4/*100%*/, discount_price,
        collateral_itr->sym);

    print_f("pledge amount: %, bid fund: %, discount price: %\n", pledge_amount,
        bid_fund, discount_price);
    asset bid_pledge = asset(pledge_amount, collateral_itr->sym);

    if (bid_pledge.amount == 0) {
        bid_pledge.amount = 1;
    }

    asset new_remain_issue = auction_itr->remain_issue;
    asset new_remain_pledge = auction_itr->remain_pledge;

    bool auctions_status = false;
    auto transfer_next_action = false;
    if (bid_pledge.amount > new_remain_pledge.amount) {

        asset lack = bid_pledge - new_remain_pledge;
        inline_transfer(collateral_itr->contract, USNBACKUP_ACCOUNT, _self, lack, string
            ("depositsys"));
        transfer_next_action = true;
        new_remain_pledge.amount = 0;
    } else {

        new_remain_pledge.amount -= bid_pledge.amount;
    }

    new_remain_issue = auction_itr->remain_issue - bid_fund;
```

Listing 3.5: usn::bid()

Recommendation It is recommended to check that the `collateral_id` declared by the bidding user is the same as the `collateral_id` being auctioned.

Status The issue has been fixed in v3 code (`61f1bbb3`).

3.5 Other Suggestions

By design of EOSIO, the permissions and usage of ordinary accounts and contracts are the same. The ordinary account will become the contract after calling the `setcode` of the system contract eosio. Moreover, the contract can call `setcode` at any time to change the execution logic. Therefore, it is necessary to pay special attention to the `setcode` action of the contract and confirm whether it meets expectations in time.

In addition, we strongly suggest to pay attention if there are a large number of token assets stored in the contract. Generally, the contract can directly transfer all the assets in the contract. Therefore, it is recommended to use a multi-signature scheme to manage account permissions for the contracts and the accounts with funds.

4 Conclusion

In this audit, we have analyzed the Defibox USN contract design and implementation. The Defibox USN is a decentralized stable coin system based on EOS. It allows users to stake tokens to generate stable coin USN, which is anchored to USD. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

As a final precaution, 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 Vulnerability Detection

5.1.1 Uncontrolled Apply Call

- Description: The security of the apply verification at the contract call entrance.
- Result: Not found
- Severity: Critical

5.1.2 Missed Permission Check

- Description: Permission check for external callable functions.
- Result: Not found
- Severity: Critical

5.1.3 Fake Token

- Description: Whether the contract has vulnerabilities against fake transfer attacks.
- Result: Not found
- Severity: Critical

5.1.4 Fake Transfer Notice

- Description: Whether the contract has fake transfer notification vulnerabilities.
- Result: Not found
- Severity: Critical

5.1.5 Transaction Rollback

- Description: Whether the contract has transaction rollback vulnerabilities.
- Result: Not found
- Severity: Critical

5.1.6 Transaction Congestion

- Description: Whether the contract has transaction congestion vulnerabilities.
- Result: Not found
- Severity: Critical

5.1.7 soft_fail State

- Description: Whether the contract can correctly identify the soft_fail status.
- Result: Not found
- Severity: Critical

5.1.8 hard_fail State

- Description: Whether the contract can correctly identify the hard_fail status.
- Result: Not found
- Severity: Critical

5.1.9 Abnormal Memo

- Description: Whether the contract can parse memo correctly.
- Result: Not found
- Severity: Medium

5.1.10 Abnormal Resource Consumption

- Description: Whether the contract has abnormal resource consumption.
- Result: Not found
- Severity: Low

References

[1] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

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

[3] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

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

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

[6] OWASP. Risk Rating Methodology. https://owasp.org/www-community/OWASP_Risk_Rating_Methodology.

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