SMART CONTRACT AUDIT REPORT

for

Koran IDO

Prepared By: Yiqun Chen

PeckShield

July 22, 2021

Document Properties

Client	PIVOTAL GLOBAL TECHNOLOGY PTE. LTD.
Title	Smart Contract Audit Report
Target	Koran IDO
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 22, 2021	Jing Wang	Final Release
1.0-rc2	July 21, 2021	Jing Wang	Release Candidate #2
1.0-rc1	July 12, 2021	Jing Wang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction **4**
- 1.1 About Koran IDO . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 8

2 Findings **10**
- 2.1 Summary . . . 10
- 2.2 Key Findings . . . 11

3 Detailed Results **12**
- 3.1 Trust Issue of Admin Keys . . . 12
- 3.2 Creation And Initialization of An IDO Outside From Factory . . . 14
- 3.3 Improved Validation Of Function Arguments . . . 16
- 3.4 Improved Token Balance Check in OptionalIDO::subscribe() . . . 17

4 Conclusion **20**

References **21**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Koran IDO protocol, 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 Koran IDO

Koran IDO is a decentralized liquidity crowd-funding platform on Ethereum. The platform provides a completed low threshold, less cost, fast and convenient way for fundraising and investment. User could create and participate into an IDO via the platform. There are three modes of IDOs, average mode, optional mode and snap up mode. All funds are protected by Ethereum smart contracts. There is an admin role to maintain the configuration of the platform.

The basic information of the Koran IDO protocol is as follows:

Table 1.1: Basic Information of The Koran IDO Protocol

Item	Description
Name	PIVOTAL GLOBAL TECHNOLOGY PTE. LTD.
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 22, 2021

In the following, we list the reviewed files and the md5 hash values used in this audit.

- MD5 (BasicIDO.sol) = 6d7939f5d1f45c81eec92d72babee1d6

- MD5 (KoranIDOFactory.sol) = a9d4b1d611ecd43aea9b558ef912fe22
- MD5 (OptionalIDO.sol) = 9c5be43b0a794a6dd1aaf2218287ea82
- MD5 (SnapUpIDO.sol) = 8ab0bb74e4e66179254cf340ce36a7ac
- MD5 (IIDO.sol) = ddf09e4078ba38bb9b28e2800375d9e8
- MD5 (IStruct.sol) = d371d558eada2e556af2ae42e31074b0

And here are the md5 hash values of the files after all fixes for the issues found in the audit have been checked in:

- MD5 (BasicIDO.sol) = b4f2d7a29660dc4de4df34dc32fe66ed
- MD5 (KoranIDOFactory.sol) = 300b3e2cc511402f05bde123b2456893
- MD5 (OptionalIDO.sol) = 29b0561c696f56898cf840f23c3af988
- MD5 (SnapUpIDO.sol) = ad7181f4bd5572a61ecca9f3b61ec8cf
- MD5 (IIDO.sol) = 09e463826727dfd3e546470c4bea7e51
- MD5 (IStruct.sol) = f534179bd9e525e0083ebebe0017b18c

In the following, we show the md5 hash value of the zip file used in this audit:

1.2 About PeckShield

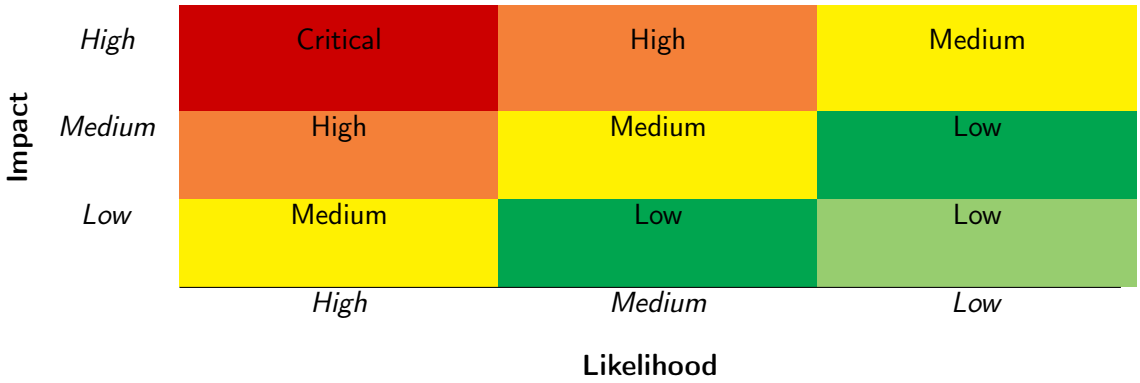
PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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;

Table 1.2: Vulnerability Severity Classification



- Severity demonstrates the overall criticality of the risk.

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

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

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.

Table 1.3: The Full List of Check Items

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this 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) or blockchain software, 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 Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Koran IDO 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	1	
Medium	1	
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 1 high-severity vulnerability, 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Koran IDO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue of Admin Keys	Security Features	Fixed
PVE-002	High	Creation And Initialization of An IDO Outside From Factory	Business Logic	Confirmed
PVE-003	Informational	Improved Validation Of Function Arguments	Coding Practices	Confirmed
PVE-004	Low	Improved Token Balance Check in OptionalIDO::subscribe()	Business Logic	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. Please refer to Section 3 for details.

3 Detailed Results

3.1 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Koran IDO protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the system-wide operations (e.g., deprecating IDOs, setting various parameters, and locking current liquidity). It also has the privilege to control or govern the flow of assets managed by this protocol. 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.

To elaborate, we show below the deprecate() function in the KoranIDOFactory contract. This function allows the owner to deprecate a specific IDO by _address.

```
147     function deprecate(address _address, uint256 _index) external onlyOwner {
148         require(_address != address(0), 'Deprecate: address can not be 0x0');
149         IdoInfo storage idoInfo = idoInfos[_index];
150         require(idoInfo.idoType < 10, 'Deprecate: had been deprecated');
151         require(idoInfo.idoAddress == _address, 'Deprecate: address error');
152         IIDO(_address).deprecate();
153         idoInfo.idoType += 10;
154         emit Deprecate(_address);
155     }
```

Listing 3.1: KoranIDOFactory::deprecate()

Also, if we examine the setFee() function in the KoranIDOFactory contract, this function allows the owner to change the feeRate to an arbitrary value. Note that the fee is calculated from

avaliableRedeemAmount divided by feeRate (line 114). If the owner changes the feeRate to 1, all the avaliableRedeemAmount will be charged as fees and the user would not get any funds from redemption.

```
161     function setFee(uint256 _fee) external onlyOwner {
162         feeRate = _fee;
163     }
```

Listing 3.2: KoranIDOFactory::setFee()

```
102     function _redeem(address _address, uint256 _realCount,uint256 _totalCount) private
            lock {
103         SubscribedInfo storage subscribedInfo = subscribedInfos[_address];
104         //calc the count of redeem
105         if (_realCount > subscribedInfo.redeemCount) {
106             uint256 avaliableRedeemAmount = subscribedInfo.subscribeAmount.mul(
                    _realCount).div(_totalCount).sub(subscribedInfo.redeemedAmount);
107             if (avaliableRedeemAmount == 0)
108                 return;
109             uint256 currentRedeemTokenAmount = avaliableRedeemAmount.mul(getTokenBalance
                    (address(this))).div((ido.currencyAmount.sub(redeemedCurrencyAmount)));
110             redeemedCurrencyAmount = redeemedCurrencyAmount.add(avaliableRedeemAmount);
111             subscribedInfo.redeemedAmount = subscribedInfo.redeemedAmount.add(
                    avaliableRedeemAmount);
112             subscribedInfo.redeemCount = _realCount;
113             subscribedInfos[_address] = subscribedInfo;
114             uint256 fee = avaliableRedeemAmount.div(ido.feeRate);
115             transfer(ido.currency, dev, fee);
116             TransferHelper.safeTransfer(ido.token, _address, currentRedeemTokenAmount);
117             transfer(ido.currency, creator, avaliableRedeemAmount.sub(fee));
118             redeemInfo[0]++;
119             emit Redeem(_address, avaliableRedeemAmount, currentRedeemTokenAmount);
120     }
```

Listing 3.3: OptionalIDO::_redeem()

Moreover, if we examine the setLock() function in OptionalIDO and SnapUpIDO, this function allows the admin to set the value of unlocked. Since the _redeem() function is guarded with the modifier lock, if theunlocked value is switched to false, all liquidity of an IDO could be locked into the contract even the redemption condition is met.

```
172     function setLock(bool _unlocked) override public onlyOwner {
173         unlocked = _unlocked;
174     }
```

Listing 3.4: OptionalIDO::setLock()

It is worrisome if the privileged owner account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 The issue has been fixed.

For the IDO deprecation privilege issue, the related functions, events and variables have been removed from the protocol.

For the feeRate issue, the team has added the requirement of require(_feeRate >= 20) to constrain the maximum of fee to 0.05. Also, the team clarifies that the feeRate can not be changed after the initialization of an IDO.

For the liquidity-lock privilege issue, the setLock() function has been adjusted to only allow the value of unlocked to be set to true, so the liquidity of an IDO could not be locked by the privileged account.

3.2 Creation And Initialization of An IDO Outside From Factory

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the KoranIDOFactory contract, the user is supposed to call the external function createIDO() to create and initialize an IDO. When the createIDO() function is called, a minFee will be charged (line 50) from the caller. However, in the OptionalIDO and SnapUpIDO contracts, the external function initialize() gives the permission of IDO Initialization to everyone. An IDO creator could instantiate an IDO instance and call the initialize() function directly to waive the minFee.

```
39      function createIDO(
40          address[] memory _tokens,
41          uint256[] memory _amounts,
42          uint[] memory _timeRanges,
43          uint256[] memory _subscribeRanges,
44          uint8[] memory _frequencyRatios,
45          string memory _url,
46          uint8 _type
47      ) public whenNotPaused payable returns (address _address){
48          require(_tokens.length == 2 && _amounts.length == 2 && _timeRanges.length == 2
                && _frequencyRatios.length == 2, "CreateIDO: the array length of tokens and
                amounts must be 2");
```

```
49          require(isContract(_tokens[0]) && currencys[_tokens[1]] && _amounts[0] > 0 &&
                _amounts[1] > 0 && _timeRanges[1].sub(now) < 90 days && _timeRanges[1] >
                _timeRanges[0] && _subscribeRanges[0] <= _subscribeRanges[1], 'CreateIDO:
                param error');
50          uint256 value = msg.value.sub(minFee);
51          ...
52      }
```

Listing 3.5: `KoranIDOFactory::createIDO()`

```
22      function initialize(IDO calldata _ido, uint8 _frequency, uint8 _ratio) override
            external {
23          require(creator == address(0));
24          require((_frequency == 1  _frequency == 7  _frequency == 30  _frequency == 90)
25              && (_ratio == 10  _ratio == 20  _ratio == 25  _ratio == 50), 'OptionalIDO:
                    params error');

27          dev = msg.sender;
28          creator = tx.origin;
29          ido = IDO({
30          url : _ido.url,
31          start : _ido.start,
32          end : _ido.end,
33          create : _ido.create,
34          token : _ido.token,
35          tokenAmount : _ido.tokenAmount,
36          currency : _ido.currency,
37          currencyAmount : _ido.currencyAmount,
38          minSubscribeAmount : _ido.minSubscribeAmount,
39          maxSubscribeAmount : _ido.maxSubscribeAmount,
40          feeRate : _ido.feeRate
41          });
42          frequency = _frequency;
43          ratio = _ratio;
44          subscribeAddresses = new address[](0);
45          redeemInfo = new uint256[](3);
46      }
```

Listing 3.6: `OptionalIDO::initialize()`

Recommendation Constrain the permission of `IDO` initialization from the `KoranIDOFactory` contract to prevent the user to create and initialize an `IDO` outside from `KoranIDOFactory`.

Status The issue has been confirmed by the team, and the team clarifies that the `IDOs` created outside from the `KoranIDOFactory` contract are not maintained by the platform.

3.3 Improved Validation Of Function Arguments

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: KoranIDOFactory
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the KoranIDOFactory contract, the createIDO() function makes several parameters validations for the input arrays before using them to create an IDO. During our analysis, we notice that in createIDO(), the validation of timeRanges could not be guaranteed by _timeRanges[1] > _timeRanges[0] (line 49). To elaborate, we show below the related code snippet of the contract.

```
39      function createIDO(
40          address[] memory _tokens,
41          uint256[] memory _amounts,
42          uint[] memory _timeRanges,
43          uint256[] memory _subscribeRanges,
44          uint8[] memory _frequencyRatios,
45          string memory _url,
46          uint8 _type
47      ) public whenNotPaused payable returns (address _address){
48          require(_tokens.length == 2 && _amounts.length == 2 && _timeRanges.length == 2
                && _frequencyRatios.length == 2, "CreateIDO: the array length of tokens and
                amounts must be 2");
49          require(isContract(_tokens[0]) && currencys[_tokens[1]] && _amounts[0] > 0 &&
                _amounts[1] > 0 && _timeRanges[1].sub(now) < 90 days && _timeRanges[1] >
                _timeRanges[0] && _subscribeRanges[0] <= _subscribeRanges[1], 'CreateIDO:
                param error');
50          ...
51      }
```

Listing 3.7: KoranIDOFactory::createIDO()

```
48  function subscribe(uint256 _subscribeAmount) override external payable lock {
49      require(!deprecated, 'OptionalIDO: had been deprecated');
50      require(now > ido.start && now < ido.end, 'not in time');
51      ...
52  }
```

Listing 3.8: KoranIDOFactory::subscribe()

Apparently, the time range requirement now > ido.start && now < ido.end from KoranIDOFactory::subscribe() (line 50) would be violated if an IDO is created with a time range from _timeRanges[0] to _timeRanges[0]+1.

Recommendation Improve the time range check from `KoranIDOFactory::createIDO()` to make sure the time range of the created `IDO` is valid.

Status The issue has been confirmed by the team.

3.4 Improved Token Balance Check in OptionalIDO::subscribe()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Koran IDO` protocol provides a `createIDO()` function for users to create an `IDO` for token-based fund-raising. Meanwhile, the protocol also provides the `subscribe()` function for users to exchange tokens by funds. However, during our analysis of these two functions, we notice the lack of token balance check of both functions may lead user to subscribe for nothing. To elaborate, we show below the implementation of `createIDO()` and `subscribe()` functions.

```
39      function createIDO(
40          address[] memory _tokens,
41          uint256[] memory _amounts,
42          uint[] memory _timeRanges,
43          uint256[] memory _subscribeRanges,
44          uint8[] memory _frequencyRatios,
45          string memory _url,
46          uint8 _type
47      ) public whenNotPaused payable returns (address _address){
48          require(_tokens.length == 2 && _amounts.length == 2 && _timeRanges.length == 2
                && _frequencyRatios.length == 2, "CreateIDO: the array length of tokens and
                amounts must be 2");
49          require(isContract(_tokens[0]) && currencys[_tokens[1]] && _amounts[0] > 0 &&
                _amounts[1] > 0 && _timeRanges[1].sub(now) < 90 days && _timeRanges[1] >
                _timeRanges[0] && _subscribeRanges[0] <= _subscribeRanges[1], 'CreateIDO:
                param error');
50          uint256 value = msg.value.sub(minFee);
51          require(value == 0  value >= minBid);
52          IERC20 erc20 = IERC20(_tokens[0]);
53          require(bytes(erc20.symbol()).length != 0 && bytes(erc20.name()).length != 0, "
                CreateIDO: token must have symbol and name");
54          bytes memory bytecode;
55          if (_type == 3) {
56              bytecode = type(SnapUpIDO).creationCode;
57          } else {
58              bytecode = type(OptionalIDO).creationCode;
```

```
59              if (_subscribeRanges[0] == _subscribeRanges[1])
60                  _type = 1;
61              else
62                  _type = 2;
63          }
64          ...
65      }
```

Listing 3.9: KoranIDOFactory::createIDO()

```
48      function subscribe(uint256 _subscribeAmount) override external payable lock {
49          require(!deprecated, 'OptionalIDO: had been deprecated');
50          require(now > ido.start && now < ido.end, 'not in time');
51          address sender = msg.sender;
52          if (ido.currency == address(0)) {
53              _subscribeAmount = msg.value;
54          }
55          require(_subscribeAmount >= ido.minSubscribeAmount && _subscribeAmount <= ido.
                maxSubscribeAmount, 'OptionalIDO: subscribe amount is error');
56          require(ido.currencyAmount.sub(raisedCurrencyAmount) >= _subscribeAmount, '
                OptionalIDO: not enough token');
57          SubscribedInfo storage subscribedInfo = subscribedInfos[sender];
58          uint256 _subscribeAmountTmp = _subscribeAmount.add(subscribedInfo.
                subscribeAmount);
59          require(_subscribeAmountTmp <= ido.maxSubscribeAmount, 'OptionalIDO: subscribe
                amount is too large');
60          if (ido.currency != address(0)) {
61              TransferHelper.safeTransferFrom(ido.currency, msg.sender, address(this),
                    _subscribeAmount);
62          }
63          raisedCurrencyAmount = raisedCurrencyAmount.add(_subscribeAmount);
64          if (subscribedInfo.subscribeAmount == 0) {
65              subscribeAddresses.push(sender);
66          }
67          subscribedInfo.subscribeAmount = _subscribeAmountTmp;
68          emit Subscribe(sender, _subscribeAmount);
69      }
```

Listing 3.10: OptionalIDO::subscribe()

We notice that, when the user creates an IDO from createIDO(), the token is transferred from the user to the IDO contract. In the meantime, the user could transfer their funds into the created IDO and exchange tokens by calling the function subscribe(). However, because the lack of token balance check on both functions, the actual token balance of IDO could be less than the tokenAmount of the IDO, (e.g., in the case of deflationary tokens). In the worst case, the user might invest their funds but get nothing.

Recommendation Improve the validations on the token balance of IDO to make sure the user won't subscribe for nothing when calling the subscribe() function.

Status The issue has been fixed by adding the validation of token balance in the `subscribe()` function.

4 Conclusion

In this audit, we have analyzed the Koran IDO protocol design and implementation. Koran IDO is a decentralized liquidity crowd-funding platform on Ethereum. User could create and participate into an IDO via the platform. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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

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

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