



SMART CONTRACT AUDIT REPORT

for

PKCO Token



Prepared By: Patrick Lou

PeckShield
March 4, 2022

Document Properties

Client	PK Crypto
Title	Smart Contract Audit Report
Target	PKCO Token
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	March 4, 2022	Luck Hu	Final Release
1.0-rc	February 11, 2022	Luck Hu	Release Candidate

Contact

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

Name	Patrick Lou
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About PKCO Token	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	8
2.1	Summary	8
2.2	Key Findings	9
3	ERC20 Compliance Checks	10
4	Detailed Results	13
4.1	Non ERC20-Compliance Of PKCO Token	13
4.2	Trust Issue Of Admin Keys	15
4.3	Proper Event Generation For Key Operations	16
5	Conclusion	18
	References	19

1 | Introduction

Given the opportunity to review the design document and related source code of the PKCO token contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of certain issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About PKCO Token

PKCO is an ERC20-compliant protocol token of PK Crypto, which is to be deployed on Binance Smart Chain (BSC). It is designed to be deflationary and non-mintable with a decreasing supply through burns. It is also a rebasing token that supports different usage scenarios: self-investment, charity donation, and prize-winning. The basic information of the audited token contracts is as follows:

Table 1.1: Basic Information of PKCO Token

Item	Description
Name	PK Crypto
Website	https://pkcrypto.org/
Type	EVM ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	March 4, 2022

In the following, we show the Git repository of reviewed contracts and the commit hash value used in this audit.

- <https://github.com/SwissArmyBud/PKCO-Token> (2986671)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- <https://github.com/SwissArmyBud/PKCO-Token> (623320e)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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 [5]:

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

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

Table 1.2: Vulnerability Severity Classification

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

We perform the audit according to the following procedures:

- 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.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- 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
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
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 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 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](#).

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.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the PKC0 token contract. 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 ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	1	
Informational	1	
Total	3	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found, and our detailed checklist can be found in Section 3. Also, though current smart contracts are well-designed and engineered, the implementation and deployment can be further improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key PKCO Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Non ERC20-Compliance Of PKCO Token	Coding Practices	Fixed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-003	Low	Proper Event Generation For Key Operations	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 4 for details.



3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `view-only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is an ERC20 inconsistency or incompatibility issue found in the audited token contracts. The detailed discussions are in Section 4.1. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` func-

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	—
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	—
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	—
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	—
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	—
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	—
	Is emitted with the from address set to <i>address(0x0)</i> when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

tions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	✓
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	✓
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	—
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

4 | Detailed Results

4.1 Non ERC20-Compliance Of PKCO Token

- ID: PVE-001
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: PKCOTokenUpgradeable/ReflectionUpgradeable
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

While examining the PKCO token implementation, we observe it presents as an ERC20-like token, but the actual implementation does not exactly follow the ERC20 specification. In the following, we examine the ERC20 compliance of PKCO token. Our analysis shows that there is an ERC20 inconsistency or incompatibility issue found in the PKCO implementation. In particular, the current `_transfer()` routine doesn't fire the `Transfer` event strictly.

To elaborate, we show below the related code snippets of this contract. The ERC20 standard specifies that the `Transfer` event MUST be triggered when tokens are transferred. However, in current `_transfer()` routine, it doesn't emit `Transfer` events for the transfer of tokens from `sender` to the Charity wallet (line 252) and the Raffle wallet (line 253). A similar issue is present in the `claimCharity()` routine where tokens are claimed from the Charity wallet to the Charity. Also, a similar issue is present in the `_claimRaffle()` routine where tokens in the raffle prize pool are claimed to the prize winners.

```
244     function _transfer(  
245         address sender,  
246         address recipient,  
247         uint256 tAmount  
248     ) internal virtual {  
249         ...  
250         uint256 rSplit = reflectionFromToken(tAmount) / 4;  
251     }
```

```

252     _raffleBalance += rSplit;
253     _charityBalance += rSplit;
254     _rOwned[sender] -= 2 * rSplit;
255
256     _burn(sender, rSplit);
257     _transferStandard(sender, recipient, rSplit);
258
259     emit Transfer(sender, recipient, tokenFromReflection(rSplit));
260     _registerForRaffle(recipient);
261 }

```

Listing 4.1: PKCOTokenUpgradeable::_transfer()

```

329     function claimCharity() public override {
330         _rOwned[_charityAddress] += _charityBalance;
331         _charityBalance = 0;
332     }

```

Listing 4.2: PKCOTokenUpgradeable::claimCharity()

```

342     function _claimRaffle(uint256 entry, address account) internal override {
343         ...
344         // Refuse to overflow total Raffle claims
345         uint256 claimedAmount = raffle._maxClaim;
346         if(claimedAmount > remainingCredit){ claimedAmount = remainingCredit; }
347
348         // Assign the entire claim and update the Raffle
349         _rOwned[account] += claimedAmount;
350         raffle._claimedAmount += claimedAmount;
351
352         _claimedRaffle[entry][account] = true;
353     }

```

Listing 4.3: PKCOTokenUpgradeable::_claimRaffle()

Our analysis also shows that, the `Transfer` event are emitted twice for the tokens transfer to the recipient. To elaborate, we show below the related code snippet of the `_transferStandard()` routine where the `Transfer` event is emitted for the first time (line 73). Following this, the `Transfer` event is emitted again in the `_transfer()` routine (line 259). It is suggested to keep only one `Transfer` event emitted for the tokens transfer to the recipient.

```

66     function _transferStandard(address sender, address recipient, uint256 rAmount)
        internal {
67         uint256 feeAmount = rAmount / _feeDivisor;
68
69         _rOwned[sender] -= rAmount;
70         _rOwned[recipient] += (rAmount - feeAmount);
71
72         _reflectFee(feeAmount, tokenFromReflection(feeAmount));
73         emit Transfer(sender, recipient, tokenFromReflection(rAmount));

```

74

}

Listing 4.4: ReflectionUpgradeable::_transferStandard()

What's more, in the PKC0 token implementation, there is one `decimals()` routine that returns the number of decimals the token uses. In the `decimals()` routine, it opts for a well known constant value of 18, imitating the relationship between [Ether](#) and [Wei](#). However, it comes to our attention that the PKC0 token contract also defines a dedicated state variable `_decimals` which is of type `uint256`. The `_decimals` variable could be initialized by a parameter (may not be of value 18) in the `initialize()` routine. In order to keep the consistency between `_decimals` variable and `decimals()` routine, it is suggested to use type `uint8` for variable `_decimals` and return `_decimals` in `decimals()` routine.

```

113     function decimals() public view virtual override returns (uint8) {
114         return 18;
115     }

```

Listing 4.5: PKC0TokenUpgradeable::decimals()

Recommendation Revise the PKC0 token implementation to ensure its ERC20-compliance.

Status This issue has been fixed in the following commits: [ef6c363](#), [e154010](#) and [623320e](#).

4.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: [RaffleUpgradeable](#), [CharityUpgradeable](#)
- Category: Security Features [\[3\]](#)
- CWE subcategory: CWE-287 [\[2\]](#)

Description

In PKC0 protocol contract, there is a privileged `owner` account (assigned in the `initialize()` routine) that plays a critical role in governing and regulating the protocol-wide operations. To elaborate, we show below the sensitive operations that are related to `owner`. Specifically, it has the authority to reset the `_charityAddress` and force the `Raffles` to be claimed to the specified account.

It would be worrisome if the `owner` account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

```

27     function resetCharity(address address_ , string memory name_) public onlyOwner {
28         claimCharity();
29         _charityName = name_;
30         _charityAddress = address_;
31     }

```

Listing 4.6: CharityUpgradeable:: resetCharity ()

```

91     function forceClaimRaffles(uint256[] memory entries , address account) public
        onlyOwner {
92         for(uint i = 0; i < entries.length; i++){
93             _claimRaffle(i, account);
94         }
95     }

```

Listing 4.7: RaffleUpgradeable:: forceClaimRaffles ()

Recommendation Promptly transfer the `owner` privilege of `PKC0` token to the intended governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team.

4.3 Proper Event Generation For Key Operations

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RaffleUpgradeable/CharityUpgradeable
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `CharityUpgradeable` contract as an example. While examining the events that reflect the `_charityAddress` dynamics, we notice there is a lack of emitting an event to reflect the `_charityAddress` being reset. Given this, we suggest to add a new event `ResetCharity()` (line 25) to be emitted in `resetCharity()` (line 31) when the `_charityAddress` is reset.


```
25  event ResetCharity(address addressOld_, address addressNew_);
26
27  function resetCharity(address address_, string memory name_) public onlyOwner {
28      claimCharity();
29      _charityName = name_;
30      _charityAddress = address_;
31      emit ResetCharity(address_, _charityAddress);
32  }
```

Listing 4.8: CharityUpgradeable::resetCharity()

We also notice there is a lack of emitting an event to reflect the Raffles dynamics. Given this, it is suggested to add proper events emitting for the key Raffles operations, e.g., `runRaffle()`, `forceClaimRaffles()` and `closeRaffle()`.

Recommendation Properly emit the above-mentioned events to timely reflect the state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been fixed in the following commit: 4288c69.



5 | Conclusion

In this security audit, we have examined the design and implementation of the PKC0 token contract. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, although no critical or high level vulnerabilities were discovered, we identified three issues of varying severities. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.



References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [4] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [5] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [6] PeckShield. PeckShield Inc. <https://www.peckshield.com>.