SMART CONTRACT AUDIT REPORT

for

EG Token

Prepared By: Xiaomi Huang

PeckShield

January 13, 2023

Document Properties

Client	EG Token
Title	Smart Contract Audit Report
Target	EG Token
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	January 13, 2023	Jing Wang	Final Release
1.0-rc	January 11, 2023	Jing Wang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the EG 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 EG Token

EG Token (EG) is a community-governed token that creates rewarding incentives for furthering the mission of social impact with crypto. The EG token contract accumulates 5% fee from every buy transaction and 5% fee from every sell transaction. No fee is collected from transfers. This 10% fee is distributed to the following 5 wallets, i.e., marketing wallet, liquidity wallet, tech wallet, donations wallet, and staking rewards wallet. The basic information of the audited EG contract is as follows:

Table 1.1: Basic Information of EG Token

Item	Description
Name	EG Token
Type	ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	January 13, 2023

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

- https://github.com/EG-Ecosystem/eg-token (3835d1f)

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

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 EG 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	0	
Low	1	■
Informational	1	■
Total	2	

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 any identified issue(s) 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 of each of them 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. However, the smart contract implementation can be improved because of the existence of 1 low-severity vulnerability and 1 informational suggestion.

Table 2.1: Key EG Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited Logic of _validateIfLiquidityAdd()	Business Logic	Confirmed
PVE-002	Low	Trust Issue Of Admin Roles	Security Features	Confirmed

Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

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. Specifically, zero amount transfers are not allowed and related events are not fired. In the surrounding two tables, we outline the respective list of basic `view-only` functions

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 $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

(Table 3.1) and key state-changing functions (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 Revisited Logic of _validateIfLiquidityAdd()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: EG
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The EG token contract designs a way to validate if users other than the owner are adding liquidity into the pair. To implement this design, the _validateIfLiquidityAdd() routine was added, and it is invoked every time when the balanceOf() routine is called. The criteria for determining a liquidity adding is if the balance being checked is from the same address that conducted the previous _transfer() in the same transaction. However, our analysis shows that there is a way to bypass the check. To elaborate, we show below the related routines.

```
205     function balanceOf(address account)
206       external
207       view
208       override
209       returns (uint256)
210     {
211       uint256 balance0 = _balanceOf(account);
212       if (
213           _lastTransfer.blockNumber == uint32(block.number) &&
214           account == _lastTransfer.to
215       ) {
216           // Balance being checked is the same address that did the last _transfer_in
217           // check if likely same transaction. If True, then it is a Liquidity Add
218           _validateIfLiquidityAdd(account, uint112(balance0));
219       }
220
221       return balance0;
222     }
```

```
223
224
225
226 function _validateIfLiquidityAdd(address account, uint112 balance0)
227   private
228   view
229 {
230   // using the data recorded in _transfer
231   if (_lastTransfer.origin == tx.origin) {
232       // May be same transaction as _transfer, check LP balances
233       address token1 = account.token1();
234
235       if (token1 == address(this)) {
236           // Switch token so token1 is always on the other side of pair
237           token1 = account.token0();
238       }
239
240       // Not LP pair
241       if (token1 == address(0)) return;
242
243       uint112 balance1 = uint112(IERC20(token1).balanceOf(account));
244
245       if (
246           balance0 > _lastTransfer.balance0 &&
247           balance1 > _lastTransfer.balance1
248       ) {
249           // Both pair balances have increased, this is a Liquidity Add
250           require(false, "EG: Liquidity can be added by the owner only");
251       } else if (
252           balance0 < _lastTransfer.balance0 &&
253           balance1 < _lastTransfer.balance1
254       ) {
255           // Both pair balances have decreased, this is a Liquidity Remove
256           require(
257               false,
258               "EG: Liquidity can be removed by the owner only"
259           );
260       }
261   }
262 }
```

Listing 4.1: `EG::_validateIfLiquidityAdd()`

This `_validateIfLiquidityAdd()` routine is used to ensure that only the owner can add liquidity. However, it is based on the assumption that when adding liquidity, the transfer of tokens into the pair and the call to `addLiquidity()` must happen in the same translation, thus same `tx.origin`. If a bad actor uses account A to transfer tokens into the pair and account B to call the `addLiquidity()` routine, then liquidity can be added even not from the owner.

Recommendation Revisit the logic for `_validateIfLiquidityAdd()` to make sure all the cases

are handled properly.

Status This issue has been confirmed. And the team clarifies that the design is mainly for user protection: if someone manages to add liquidity, it will not be an issue or a risk to the project.

4.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: EG
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the EG token protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the protocol-wide operations (e.g., implementation replacement and parameters configuration). 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 owner account and its related privileged accesses in current contract.

To elaborate, we show the related routines from the EG token contract. These routines allow the owner account to set the implementation of the token and add blacklist accounts.

```
19      function setImplementation(address implementation_) external onlyOwner {
20          StorageSlot.setAddressAt(_IMPL_SLOT, implementation_);
21      }
22
23      function addClientsToBlackList(address[] calldata accounts)
24          external
25          onlyOwner
26      {
27          for (uint256 i; i < accounts.length; i++) {
28              require(
29                  accounts[i] != address(0),
30                  "EG: Zero address can't be added to blacklist"
31              );
32          }
33
34          for (uint256 i; i < accounts.length; i++) {
35              if (!blackList[accounts[i]]) {
36                  blackList[accounts[i]] = true;
37              }
38          }
39
```

```
40          emit AddClientsToBlackList(accounts);
41      }
```

Listing 4.2: EG Token Contract

We understand the need of the privileged functions for contract maintenance, but 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 This issue has been confirmed. The team clarifies the `owner` account will be assigned to a multi-sig contract.

5 Conclusion

In this security audit, we have examined the design and implementation of the EG 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 one informational suggestion and one low-severity issue. 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.