



SMART CONTRACT AUDIT REPORT

for

CUSD Token



Prepared By: Patrick Liu

PeckShield
March 4, 2022

Document Properties

Client	Coin98
Title	Smart Contract Audit Report
Target	CUSD Token
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	March 4, 2022	Jing Wang	Final Release
1.0-rc	March 2, 2022	Jing Wang	Release Candidate

Contact

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

Name	Patrick Liu
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About CUSD 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	Trust Issue Of Admin Roles	13
4.2	Constant/Immutable States If Fixed Or Set at Constructor()	15
4.3	Safe-Version Replacement With safeTransfer()	16
5	Conclusion	19
	References	20

1 | Introduction

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

`CUSD Token` is an ERC20-compliant stablecoin that is closely related to the `Coin98` protocol's contract in minting tokens. The main functionality includes full ERC20 compatibility with additional extensions that are designed to mint a corresponding number of `CUSD` tokens based on market price of `Coin98` tokens.

The basic information of `CUSD Token` is as follows:

Table 1.1: Basic Information of `CUSD Token`

Item	Description
Name	Coin98
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	March 4, 2022

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

- <https://github.com/coin98/coin98-eco-currency-contract.git> (bd95503)

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

- <https://github.com/coin98/coin98-eco-currency-contract.git> (09c4a33)

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 `CUSD` 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. However, the smart contract implementation can be improved because of the existence of 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendations.

Table 2.1: Key CUSD Token Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue Of Admin Roles	Security Features	Confirmed
PVE-002	Informational	Constant/Immutable States If Fixed Or Set at Constructor()	Coding Practices	Fixed
PVE-003	Low	Safe-Version Replacement With safe-Transfer()	Coding Practices	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 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 no ERC20 inconsistency or incompatibility issue found in the audited CUSD Token. In the surrounding two tables, we outline the respective list of basic [view-only](#) functions (Table [3.1](#)) and key [state-changing](#) functions (Table [3.2](#)) according to the widely-

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()	✓

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 <code>transfer()/transferFrom()</code> calls	—
Rebasing	The <code>balanceOf()</code> 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 users to burn tokens of a specific address	✓

4 | Detailed Results

4.1 Trust Issue Of Admin Roles

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: CUSD
- Category: Security Features [3]
- CWE subcategory: CWE-287 [2]

Description

In the CUSD token contract, there is a privileged owner account (assigned in the [constructor](#)) that plays a critical role in governing and regulating the token-related operations (e.g., account blacklisting, funds withdrawing and token minting).

To elaborate, we show below the privileged functions in the CUSD contract.

```
543  /**
544   * @dev Adds account to blacklist
545   * @param account_ The address to blacklist
546   */
547  function blacklist(address account_) external onlyOwner {
548      _blacklisted[account_] = true;
549      emit Blacklisted(account_);
550  }
551
552  /**
553   * @dev Removes account from blacklist
554   * @param account_ The address to remove from the blacklist
555   */
556  function unBlacklist(address account_) external onlyOwner {
557      _blacklisted[account_] = false;
558      emit UnBlacklisted(account_);
559  }
```

Listing 4.1: CUSD::blacklist() and unBlacklist()

```

570 /// @dev withdraw token from contract
571 /// @param token_ address of the token, use address(0) to withdraw gas token
572 /// @param destination_ recipient address to receive the fund
573 /// @param amount_ amount of fund to withdraw
574 function withdraw(address token_, address destination_, uint256 amount_) external
    onlyOwner {
575     require(destination_ != address(0), "Withdrawable: Destination is zero address");
576
577     uint256 availableAmount;
578     if(token_ == address(0)) {
579         availableAmount = address(this).balance;
580     } else {
581         availableAmount = IERC20(token_).balanceOf(address(this));
582     }
583
584     require(amount_ <= availableAmount, "Withdrawable: Not enough balance");
585
586     if(token_ == address(0)) {
587         destination_.call{value:amount_}("");
588     } else {
589         IERC20(token_).transfer(destination_, amount_);
590     }
591
592     emit Withdrawn(_msgSender(), destination_, token_, amount_);
593 }
594
595 /// @dev withdraw NFT from contract
596 /// @param token_ address of the token, use address(0) to withdraw gas token
597 /// @param destination_ recipient address to receive the fund
598 /// @param tokenId_ ID of NFT to withdraw
599 function withdrawNft(address token_, address destination_, uint256 tokenId_) external
    onlyOwner {
600     require(destination_ != address(0), "Withdrawable: destination is zero address");
601
602     IERC721(token_).transferFrom(address(this), destination_, tokenId_);
603
604     emit Withdrawn(_msgSender(), destination_, token_, 1);
605 }

```

Listing 4.2: CUSD::withdraw() and withdrawNft()

```

885 function setMinter(address newMinter) public onlyOwner {
886     address oldMinter = _minter;
887     _minter = newMinter;
888     emit MinterUpdated(oldMinter, newMinter);
889 }
890
891 function mint(address account, uint256 amount) public
892     onlyMinter
893     notBlacklisted(_msgSender())
894     notBlacklisted(account)
895 {
896     _mint(account, amount);

```

897 }

Listing 4.3: `CUSD::setMinter()` and `mint()`

We understand the need of the privileged functions for contract upgrade, but at the same time the extra power to the admin roles may also be a counter-party risk to the contract users. 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. Also list `keeper` accounts granted by `owner` explicitly to users.

Status This issue has been confirmed by the teams. And the team clarifies a multi-sig contract will be assigned to be owner of the contract after deployment.

4.2 Constant/Immutable States If Fixed Or Set at Constructor()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CUSD`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

Since version 0.6.5, [Solidity](#) introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., “a constant, once assigned in the constructor, is read-only during the subsequent operation.”

In the following, we show the key state variables defined in CUSD. If there is no need to dynamically update these key state variables, e.g., `_name` and `_symbol`, they can be declared as `immutable` for gas efficiency.

In addition, we notice the state variable `_decimals` is a constant and we can simply define it as a `constant` to avoid gas cost for the access.

```

632     contract CUSD is Context, Ownable, Pausable, Blacklistable, Withdrawable, IERC20 {
633         ...
634         string private _name;
635         string private _symbol;
636         uint8 private _decimals;
637         ..
638     }
```

Listing 4.4: CUSD.sol

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status This issue has been addressed in the following commit: 09c4a33.

4.3 Safe-Version Replacement With `safeTransfer()`

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

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below.

```

121     /**
122     * @dev transfer token for a specified address
123     * @param _to The address to transfer to.
124     * @param _value The amount to be transferred.
125     */
126     function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
127         uint fee = (_value.mul(basisPointsRate)).div(10000);
128         if (fee > maximumFee) {
```



```

129         fee = maximumFee;
130     }
131     uint sendAmount = _value.sub(fee);
132     balances[msg.sender] = balances[msg.sender].sub(_value);
133     balances[_to] = balances[_to].add(sendAmount);
134     if (fee > 0) {
135         balances[owner] = balances[owner].add(fee);
136         Transfer(msg.sender, owner, fee);
137     }
138     Transfer(msg.sender, _to, sendAmount);
139 }

```

Listing 4.5: USDT Token Contract

It is important to note the `transfer()` function does not have a return value. However, the IERC20 interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address to, uint tokens) virtual public returns (bool success)`. As a result, the call to `transfer()` may expect a return value. With the lack of return value of USDT's `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `withdraw()` routine in the CUSD contract. If USDT is given as `token_`, the unsafe version of `IERC20(token_).transfer(destination_, amount_)` (line 589) may revert as there is no return value in the USDT token contract's `transfer()` implementation (but the IERC20 interface expects a return value)!

```

570     /// @dev withdraw token from contract
571     /// @param token_ address of the token, use address(0) to withdraw gas token
572     /// @param destination_ recipient address to receive the fund
573     /// @param amount_ amount of fund to withdraw
574     function withdraw(address token_, address destination_, uint256 amount_) external
575         onlyOwner {
576         require(destination_ != address(0), "Withdrawable: Destination is zero address");
577
578         uint256 availableAmount;
579         if(token_ == address(0)) {
580             availableAmount = address(this).balance;
581         } else {
582             availableAmount = IERC20(token_).balanceOf(address(this));
583         }
584         require(amount_ <= availableAmount, "Withdrawable: Not enough balance");
585
586         if(token_ == address(0)) {
587             destination_.call{value:amount_}("");

```

```
588     } else {  
589         IERC20(token_).transfer(destination_, amount_);  
590     }  
591  
592     emit Withdrawn(_msgSender(), destination_, token_, amount_);  
593 }
```

Listing 4.6: CUSD::withdraw()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()`.

Status This issue has been fixed in the commit: [1a29562](#).



5 | Conclusion

In this security audit, we have examined the design and implementation of the `CUSD` 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 that were promptly confirmed and addressed by the team. 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>.