SMART CONTRACT AUDIT REPORT

for

WUSD

Prepared By: Xiaomi Huang

PeckShield

April 23, 2023

Document Properties

Client	WUSD
Title	Smart Contract Audit Report
Target	WUSD
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	April 23, 2023	Xuxian Jiang	Final Release
1.0-rc	April 6, 2023	Xuxian Jiang	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 smart contract source code of the WUSD 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 WUSD

`Wrapped USD` is a governance-free, immutable, multi-fiatcoin wrapper for the 6 `USD fiatcoins` -`USDT`, `USDC`, `BUSD`, `USDP`, `TUSD`, and `GUSD`. The incentive token, `GLOVE`, represents the first, live implementation of a new kind of token: a `utility credit` token. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of WUSD

Item	Description
Name	WUSD
Website	https://wusd.fi/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	April 23, 2023

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

- https://github.com/witbub/wusd-peckshield.git (45ec073)

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).

Table 1.2: Vulnerability Severity Classification

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

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;
- 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

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

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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 WUSD 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

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

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 3 low-severity vulnerabilities.

Table 2.1: Key WUSD Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited deregister() Logic in Frontender	Coding Practices	Resolved
PVE-002	Low	Improved _transferCredits() Logic in Glove	Business Logic	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Confirmed

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 3 for details.

3 | Detailed Results

3.1 Revisited deregister() Logic in Frontender

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Frontender`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `WUSD` protocol has a `Frontender` contract, which is designed to keep track of active frontend referrals. While examining the related referrer registration/de-registration logic, we notice current referrer de-registration implementation needs to be revised.

To elaborate, we show below the related `deregister()` routine. As the name indicates, this routine is used to de-register a current referrer. It comes to our attention that the credit calculation of the de-registered referrer needs to be computed as `_percent(creditless, Math.min((_referred[msg.sender] / 100_000e18)* 100_00, 100_00))` instead of the current `_percent(creditless, Math.min((_referred[msg.sender] / 100_000e18)* 100, 100_00))` (line 118). The reason is the intended 100% is encoded as 100_00, not 100.

```
103   function deregister () external nonReentrant
104   {
105     _isUnwrapped();
106     require(_registered(msg.sender), "!registered");


109     uint256 creditless = IGlove(_GLOVE).creditlessOf(msg.sender);


112     /**
113      * made equivalent to epoch size in Glove.sol where epoch = 100K units
114      *
115      *
```

```
   * simply: min(((referrals / 100K) * 100)%, 100%); % is in basis point in code
   */
  uint256 credits = _percent(creditless, Math.min((_referred[msg.sender] / 100_000e18)
      * 100, 100_00));


  _referred[msg.sender] = 0;

  // burns any uncredited GLO
  IGlove(_GLOVE).burn(msg.sender, creditless - credits);
  IGlove(_GLOVE).creditize(msg.sender, credits);


  emit Deregister(msg.sender);
}
```

Listing 3.1: `Frontender::deregister()`

Recommendation Revise the above routine to properly compute the credits of the de-registered referrer.

Status The issue has been resolved as the team confirms it is part of the design.

3.2 Improved _transferCredits() Logic in Glove

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Glove`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The utility credit token `Glove` is the incentive token that can be earned for wrapping and running a frontend. In the process of examining the credit-transfering logic, we notice the current implementation can be improved.

Specifically, we show below the related `_transferCredits()` routine. It has a specific design in adjusting the credit balance of both sender and receiver. In particular, if both sender and receiver are not creditors, there is a further check `if (from == tx.origin)` (line 164), which considers ‘`sender`‘ = ‘`from`‘ = `msg.sender` (line 164). Note that the statement of ‘`from`‘ = `msg.sender` may not always hold in the `transferFrom()` case.

```
145  function _transferCredits (address from, address to, uint256 amount) internal
146  {
147    uint256 credit = _credit[from]; // credit of sender
```

```
    bool senderIsCreditor = hasRole(CREDITOR_ROLE, from);
    bool recipientIsCreditor = hasRole(CREDITOR_ROLE, to);


    if (!senderIsCreditor && !recipientIsCreditor)
    {
      require(credit >= amount, "GLO: amount > credit");
    }

    _credit[from] = credit > amount ? credit - amount : 0;

    if (senderIsCreditor  recipientIsCreditor)
    {
      _credit[to] += amount;
    }
    else if (from == tx.origin)
    {
      _credit[to] += (_balance[to] > amount ? amount : ((amount * 99_00) / 100_00));
    }
  }
```

Listing 3.2: `Glove::_transferCredits()`

Recommendation Revisit the above routine to ensure the EOA-related check is part of the intended design.

Status The issue has been resolved as the team confirms it is part of the design.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Glove`, `Frontender`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `WUSD` protocol, there is a privileged account with the `DEFAULT_ADMIN_ROLE` role that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and role assignment). 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.

```
  // mints `amount` tokens to `account` with full credit
```

```
function mint (address account, uint256 amount) external
{
  _mint(account, amount);
  _creditize(account, amount);
}

// mints `amount` tokens to `account` but with 0 credit
function mintCreditless (address account, uint256 amount) external
{
  _isCreditor();


  _mint(account, amount);
}

// grants `amount` credit to `account`
function creditize (address account, uint256 amount) external returns (bool)
{
  _creditize(account, amount);


  return true;
}

// seizes `amount` credit from `account`
function decreditize (address account, uint256 amount) external returns (bool)
{
  _decreditize(account, amount);


  return true;
}
```

Listing 3.3: Example Privileged Operations in the `Glove` Contract

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 plans to revoke the permission after the mainnet deployment becomes stable.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Wrapped USD` protocol, which is a governance-free, immutable, multi-fiatcoin wrapper for the 6 `USD fiatcoins` -`USDT`, `USDC`, `BUSD`, `USDP`, `TUSD`, and `GUSD`. The incentive token, `GLOVE`, represents the first, live implementation of a new kind of token: a `utility credit` token. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-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-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.