SMART CONTRACT AUDIT REPORT

for

Okse Card

Prepared By: Xiaomi Huang

PeckShield

Jun 10, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	Jun 10, 2022	Shulin Bie	Final Release
1.0-rc	May 31, 2022	Shulin Bie	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 Okse Card 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 Okse Card

Okse is a decentralized non-custodial system built to revolutionize the financial market. The Okse Card protocol, as an important component of the Okse ecosystem, allows the users to access and spend their cryptocurrency without counter-party risk in over 60 million shops worldwide. It brings great convenience to the users. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Okse Card

Item	Description
Target	Okse Card
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	Jun 10, 2022

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

- https://github.com/Okseio/oksecard-contracts.git (9972cbb)

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

- https://github.com/Okseio/oksecard-contracts.git (0fc0890)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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

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

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.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], 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 `Okse Card` 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	2	■■
Low	2	■■
Informational	1	■
Total	5	

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 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Okse Card Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Price Manipulation In Current Implementation	Time and State	Fixed
PVE-002	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Low	Suggested SafeMath Usage In OkseCard	Numeric Errors	Fixed
PVE-004	Informational	Suggested Event Generation For Key Operations	Coding Practices	Fixed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Possible Price Manipulation In Current Implementation

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `CustomPriceFeed2`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

While examining the `OkseCard` contract, we notice the current implementation can be improved with effective slippage control. To elaborate, we show below the related code snippet of the `OkseCard` contract.

By design, the internal `calculateAmount()` routine is designed to swap a certain amount of token to the exact amount of `USDT`. Within the routine, the `getAmountsIn()` routine of the `swapper` is called (lines 835-838) to calculate the amount of the input token needed, and then the `_swap()` routine of the `swapper` is called (line 848) to swap the input token to `USDT`. However, we observe it essentially does not specify any restriction on possible slippage. Given this, we suggest to specify the `amountInMax` to prevent possible front-running attacks.

```
803     function calculateAmount(
804         address market,
805         address userAddr,
806         uint256 usdAmount,
807         address targetAddress,
808         address feeAddress,
809         uint256 feePercent
810     ) internal returns (uint256 spendAmount) {
811         uint256 addFeeUsdAmount;
812         if (feeAddress != address(0)) {
813             addFeeUsdAmount =
814                 usdAmount +
815                 (usdAmount * feePercent) /
```

```
                10000 +
                buyTxFee;
        } else {
            addFeeUsdAmount = usdAmount;
        }
        // change addFeeUsdAmount to _USDT asset amounts
        // uint256 assetAmountIn = getAssetAmount(market, addFeeUsdAmount);
        // assetAmountIn = assetAmountIn + assetAmountIn / 10; //price tolerance = 10%
        uint256 usdtTotalAmount = convertUsdAmountToAssetAmount(
            addFeeUsdAmount,
            _USDT
        );
        if (market != _USDT) {
            // we need to change something here, because if there are not pair {market, _USDT} , then we have to add another path
            // so please check the path is exist and if no, please add market, weth, usdt to path
            address[] memory path = ISwapper(swapper).getOptimumPath(
                market,
                _USDT
            );
            uint256[] memory amounts = ISwapper(swapper).getAmountsIn(
                usdtTotalAmount,
                path
            );
            require(amounts[0] <= usersBalances[userAddr][market], "ua");
            usersBalances[userAddr][market] =
                usersBalances[userAddr][market] -
                amounts[0];
            TransferHelper.safeTransfer(
                path[0],
                ISwapper(swapper).GetReceiverAddress(path),
                amounts[0]
            );
            ISwapper(swapper)._swap(amounts, path, address(this));
        } else {
            require(usdtTotalAmount <= usersBalances[userAddr][market], "uat");
            usersBalances[userAddr][market] =
                usersBalances[userAddr][market] -
                usdtTotalAmount;
        }
        ...
    }
```

Listing 3.1: `OkseCard::calculateAmount()`

Additionally, the `CustomPriceFeed::getPrice()` routine is designed as price oracle. Within the routine, the `getAmountsOut()` is called (lines 114-117) to estimate the specified `token` price. However, it ignores the fact that the `token/USDT` pair may have been price-manipulated, which directly undermines the assumption of the `Okse Card` design. Note that the `CustomPriceFeed2::getPrice()` routine shares the similar issue.

```
    function getPrice() public view returns (int256) {
        address[] memory path = new address[](2);
        path[0] = token;
        path[1] = USDT;
        uint256 amountIn = testAmountsIn;
        uint256[] memory amounts = IUniswapV2Router01(router).getAmountsOut(
            amountIn,
            path
        );
        int256 _dec = _decimals +
                    ERC20Interface(token).decimals() -
                    ERC20Interface(USDT).decimals();
        int256 price;
        if (_dec >= 0) {
            price = int256((amounts[1] * uint256(10**uint256(_dec))) / testAmountsIn);
        } else {
            price = int256(amounts[1] / uint256(10**uint256(-_dec)) / testAmountsIn);
        }
        return price;
    }
```

Listing 3.2: `CustomPriceFeed::getPrice()`

Recommendation Improve the logic of above-mentioned routines to prevent possible price manipulation.

Status The issue has been addressed by the following commit: `dd13c54`.

3.2 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `OkseCard`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `Okse Card` implementation, the `OkseCard` contract is the main entry for interaction with users. In particular, one entry routine, i.e., `deposit()`, accepts the deposits of the supported assets. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `OkseCard` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
    function deposit(address market, uint256 amount)
        public
        marketEnabled(market)
        nonReentrant
        noEmergency
    {
        TransferHelper.safeTransferFrom(
            market,
            msg.sender,
            address(this),
            amount
        );
        _addUserBalance(market, msg.sender, amount);
        emit UserDeposit(msg.sender, market, amount);
    }
```

Listing 3.3: `OkseCard::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the `OkseCard` contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Okse Card`. In `Okse Card` protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status The issue has been confirmed by the team. The team decides to leave it as is considering there is no need to support deflationary/rebasing token.

3.3 Suggested SafeMath Usage In OkseCard

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OkseCard
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [1]

Description

SafeMath is a Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, we find that it is not widely used in OkseCard contract.

In particular, while examining the logic of the OkseCard contract, we notice that there are several functions without the overflow/underflow protection. In the following, we use the _addUserBalance () function as an example. In the _addUserBalance() function, it comes to our attention that the arithmetic operation (lines 540) does not use the SafeMath library to prevent overflows or underflows, which may introduce unexpected behavior. We suggest to use SafeMath to avoid unexpected overflows or underflows.

```
534     function _addUserBalance(
535         address market,
536         address userAddr,
537         uint256 amount
538     ) internal marketEnabled(market) {
539         uint256 beforeAmount = usersBalances[userAddr][market];
540         usersBalances[userAddr][market] += amount;
541         onUpdateUserBalance(
542             userAddr,
543             market,
544             usersBalances[userAddr][market],
545             beforeAmount
546         );
547     }
```

Listing 3.4: OkseCard::_addUserBalance()

Note that the routines that involve arithmetic operations can be similarly improved.

Recommendation Use SafeMath to avoid unexpected overflows or underflows.

Status The issue has been addressed by the following commit: 0fc0890.

3.4 Suggested Event Generation For Key Operations

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: OkseCard
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

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.

While examining the events that reflect the protocol dynamics, we notice there are several privileged routines that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    function setPriceOracle(address _priceOracle) public onlyOwner {
        priceOracle = _priceOracle;
    }

    // verified
    function setSwapper(address _swapper) public onlyOwner {
        swapper = _swapper;
    }

    function setDefaultMarket(address market)
        public
        marketEnabled(market)
        marketSupported(market)
        onlyOwner
    {
        defaultMarket = market;
    }
```

Listing 3.5: OkseCard

With that, we suggest to emit meaningful events in these privileged routines. Also, the key event information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being indexed.

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

Status The issue has been addressed by the following commit: `a4cd89b`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Okse Card` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters and manage the privileged `signer` account). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
    function setPriceOracle(address _priceOracle) public onlyOwner {
        priceOracle = _priceOracle;
    }

    // verified
    function setSwapper(address _swapper) public onlyOwner {
        swapper = _swapper;
    }

    function setDefaultMarket(address market)
        public
        marketEnabled(market)
        marketSupported(market)
        onlyOwner
    {
        defaultMarket = market;
    }

    // verified
    function addSigner(address _signer) public onlyGovernor {
        _addSigner(_signer);
    }

    // verified
    function removeSigner(address _signer) public onlyGovernor {
        _removeSigner(_signer);
```

```
    }
```

Listing 3.6: `OkseCard`

```
    function setDirectPrice(address asset, uint256 price) public {
        require(msg.sender == admin, "only admin can set price");
        emit PricePosted(asset, prices[asset], price, price);
        prices[asset] = price;
    }

    function setPriceFeed(address asset, address priceFeed) public {
        require(msg.sender == admin, "only admin can set price");
        emit PriceFeedChanged(asset, priceFeeds[asset], priceFeed);
        priceFeeds[asset] = priceFeed;
    }
```

Listing 3.7: `OkseCardPriceOracle`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest a multi-sig account plays the privileged `owner` account to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status This issue has been mitigated via introducing `multi-sig` mechanism in the protocol.

4 Conclusion

In this audit, we have analyzed the `Okse Card` design and implementation. `Okse` is a decentralized non-custodial system built to revolutionize the financial market. The `Okse Card` protocol, as an important component of the `Okse` ecosystem, allows the users to access and spend their cryptocurrency without counter-party risk in over 60 million shops worldwide. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[10] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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