SMART CONTRACT AUDIT REPORT

for

Djed

Prepared By: Xiaomi Huang

PeckShield

April 7, 2023

Document Properties

Client	Djed Alliance
Title	Smart Contract Audit Report
Target	Djed
Version	1.1
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	April 7, 2023	Luck Hu	Post Release #1
1.0	February 10, 2023	Luck Hu	Final Release
1.0-rc	February 6, 2023	Luck Hu	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 `Djed` 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 Djed

`Djed` is an autonomous crypto-backed stable coin protocol which maintains a reserve of the EVM-compatible blockchain's independent native currency to back the stable coin in circulation. `Djed` issues two ERC20-compliant coins: the `stablecoin` and the `reservecoin`, which can be bought/sold directly with `Djed`. The basic information of the `Djed` protocol is as follows:

Table 1.1: Basic Information of The `Djed` Protocol

Item	Description
Website	https://www.djed.one
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 7, 2023

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

- https://github.com/DjedAlliance/Djed-Solidity (70eb3880)

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

- https://github.com/DjedAlliance/Djed-Solidity (4ddb6ec)

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

Likelihood

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 `Djed` 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■ ■
Medium	0	
Low	1	■
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 2 high-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key Djed Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improved SC Price Calculation in buyStableCoins()	Business Logic	Fixed
PVE-002	High	Wrong Parameter to Calculate rcBP in buyReserveCoins()	Arguments and Parameters	Fixed
PVE-003	Low	Improved Validations in Buy/Sell of SC/RC	Error Conditions, Return Values, Status Codes	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 details.

3 | Detailed Results

3.1 Improved SC Price Calculation in buyStableCoins()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Djed`
- Category: Business Logic [?]
- CWE subcategory: CWE-841 [?]

Description

In the `Djed` protocol, users can buy the `stablecoin` from the `Djed` contract with the base coins based on the `stablecoin` price. While examining the logic to calculate the `stablecoin` price, we notice it does not correctly take the remaining payment amount to calculate the `stablecoin` price.

To elaborate, we show below the code snippets from the `Djed` contract. As the name indicates, the `buyStableCoins()` routine is used to buy the `stablecoin`. It first takes fees from the initial payment (`msg.value`) and gets the remaining payment amount (`amountBC`) that could be used to calculate the new `stablecoin` amount (line 93). The `stablecoin` price is calculated by calling the `scPrice()` routine which accepts a parameter `_currentPaymentAmount` that represents the new increased reserve amount in the contract. Because the fees (`ui` and `treasury`) have been transferred out from the contract to the recipients (lines $107-108$), so the increased reserve amount is `(msg.value - f_t -f_ui)`. However, we notice it directly uses the initial payment amount, i.e., `msg.value`, to calculate the `stablecoin` price. As a result, the calculated price may be a bit lower than expectation.

```
92    function buyStableCoins(address receiver, uint256 feeUI, address ui) external
          payable {
93        uint256 amountBC = deductFees(msg.value, feeUI, ui); // side-effect: increases `
              treasuryRevenue`
94        uint256 scP = scPrice(msg.value);
95        uint256 amountSC = (amountBC * scDecimalScalingFactor) / scP;
96        require(amountSC <= txLimit  stableCoin.totalSupply() <= thresholdSupplySC, "
              buySC: tx limit exceeded");
97        stableCoin.mint(receiver, amountSC);
```

```
        require(isRatioAboveMin(scP), "buySC: ratio below min");
        emit BoughtStableCoins(msg.sender, receiver, amountSC, msg.value);
    }

    function deductFees(uint256 value, uint256 fee_ui, address ui) internal returns (
        uint256) {
        uint256 f = (value * fee) / scalingFactor;
        uint256 f_ui = (value * fee_ui) / scalingFactor;
        uint256 f_t = (value * treasuryFee()) / scalingFactor;
        treasuryRevenue += f_t;
        payable(treasury).transfer(f_t);
        payable(ui).transfer(f_ui);
        // payable(address(this)).transfer(f); // this happens implicitly, and thus ‘f‘ is effectively transferred to the reserve.
        return value - f - f_ui - f_t; // amountBC
    }

    function scPrice(uint256 _currentPaymentAmount) public view returns (uint256);
```

Listing 3.1: `Djed::buyStableCoins()`

Recommendation Revisit the `buyStableCoins()` routine to use the correct increased reserve amount to calculate the `stablecoin` price.

Status The issue has been fixed by this commit: `b2ce5b73`.

3.2 Wrong Parameter to Calculate rcBP in buyReserveCoins()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Djed`
- Category: Arg.s and Parameters [?]
- CWE subcategory: N/A

Description

In the `Djed` protocol, users can buy the `reservecoin` based on the buy price by interacting directly with the `Djed` contract. While reviewing the logic to calculate the `reservecoin` buy price, we notice it transfers wrong parameter to the `rcBuyingPrice()` routine.

To elaborate, we show below the code snippets of the `buyReserveCoins()/rcBuyingPrice()` routines. As the name indicates, the `buyReserveCoins()` routine is used to buy the `reservecoin`. It calculates the buy price by calling `rcBuyingPrice(scP)` (line 116), where the input `scP` is the `stablecoin` price. However, it comes to our attention that there is no such `rcBuyingPrice()` routine implemented in the contract which takes only the `stablecoin` price as the parameter. Our analysis shows that there is a `function rcBuyingPrice(uint256 _scPrice, uint256 _currentPaymentAmount)` defined (line 127), which takes the `stablecoin` price as the first parameter (`_scPrice`), and the payment amount as the second parameter (`_currentPaymentAmount`). Based on this, it is suggested to replace the `rcBuyingPrice(scP)` with `rcBuyingPrice(scP, msg.value)`.

```
113
114   function buyReserveCoins(address receiver, uint256 fee_ui, address ui) external
          payable {
115     uint256 scP = scPrice(msg.value);
116     uint256 rcBP = rcBuyingPrice(scP);
117     require(msg.value <= (txLimit * scP) / scDecimalScalingFactor  stableCoin.
            totalSupply() <= thresholdSupplySC,
118         "buyRC: tx limit exceeded"
119     );
120     uint256 amountBC = deductFees(msg.value, fee_ui, ui); // side-effect: increases `
            treasuryRevenue`
121     uint256 amountRC = (amountBC * rcDecimalScalingFactor) / rcBP;
122     reserveCoin.mint(receiver, amountRC);
123     require(isRatioBelowMax(scP)  stableCoin.totalSupply() <= thresholdSupplySC, "buyRC:
             ratio above max");
124     emit BoughtReserveCoins(msg.sender, receiver, amountRC, msg.value);
125   }
126
127   function rcBuyingPrice(uint256 _scPrice, uint256 _currentPaymentAmount) internal view
          returns (uint256) {
128     return reserveCoin.totalSupply() == 0
129             ? rcMinPrice
130             : Math.max(rcTargetPrice(_scPrice, _currentPaymentAmount), rcMinPrice);
131 }
```

Listing 3.2: `Djed.sol`

Recommendation Revisit the `buyReserveCoins()` routine to calculate the `reservecoin` buy price by calling `rcBuyingPrice(msg.value)`.

Status The issue has been fixed by this commit: `5f81ad6d`.

3.3 Improved Validations in Buy/Sell of SC/RC

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Djed`
- Category: Status Codes [?]
- CWE subcategory: CWE-391 [?]

Description

In the `Djed` protocol, users can buy/sell the `stablecoin/reservecoin` tokens by directly interacting with the `Djed` contract. While reviewing the logic in the buy/sell routines of the `stablecoin/reservecoin`, we notice there is a lack of proper validation on the new token amount to be minted for a buy as well as the base token amount to be transferred to the seller for a sell.

To elaborate, we show below the code snippets of the `buyStableCoins()/sellStableCoins()` routines. As the name indicates, the `buyStableCoins()` routine is used to buy the `stablecoin`. It first charges fees from the payment (line 94), and then the remaining payment is used to calculate the new `stablecoin` amount to be minted for the buy (line 96). So it is possible that the calculated `stablecoin` amount may be 0 if the remaining payment is too little or the `stablecoin` price is too high. As a result, there may be no `stablecoin` token minted to the buyer while all the payment is used. Based on this, it is suggested to add a proper validation on the new `stablecoin` amount to ensure the buyer can get some `stablecoin` minted from the buy or revert the buy.

Similarly, the `sellStableCoins()` routine is used to sell the `stablecoin`. The amount of the base coins the seller can receive is calculated from `value = (amountSC * scP)/ scDecimalScalingFactor` (line 107) where the `amountSC` is the `stablecoin` amount to sell, and the `scP` is the `stablecoin` price. After charging fees from the `value`, the seller can receive `amountBC` of base coins. So it is possible that the seller may receive 0 base coin if the `amountSC` or the `scP` is too small, or the fee rates are too high. Based on this, it is suggested to add a proper validation for the `amountBC` to ensure the seller can receive some base coins for the sell or revert the sell.

Note the same issue is also applicable to the buy/sell of the `reservecoin`.

```
92
93   function buyStableCoins(address receiver, uint256 feeUI, address ui) external payable {
94       uint256 amountBC = deductFees(msg.value, feeUI, ui); // side-effect: increases `treasuryRevenue`
95       uint256 scP = scPrice(msg.value);
96       uint256 amountSC = (amountBC * scDecimalScalingFactor) / scP;
97       require(amountSC <= txLimit || stableCoin.totalSupply() <= thresholdSupplySC, "buySC: tx limit exceeded");
98       stableCoin.mint(receiver, amountSC);
99       require(isRatioAboveMin(scP), "buySC: ratio below min");
100      emit BoughtStableCoins(msg.sender, receiver, amountSC, msg.value);
```

```
    }

    function sellStableCoins(uint256 amountSC, address receiver, uint256 feeUI, address ui) external {
        require(stableCoin.balanceOf(msg.sender) >= amountSC, "sellSC: insufficient SC balance");
        require(amountSC <= txLimit || stableCoin.totalSupply() <= thresholdSupplySC, "sellSC: tx limit exceeded");
        uint256 scP = scPrice(0);
        uint256 value = (amountSC * scP) / scDecimalScalingFactor;
        uint256 amountBC = deductFees(value, feeUI, ui); // side-effect: increases `treasuryRevenue`
        stableCoin.burn(msg.sender, amountSC);
        payable(receiver).transfer(amountBC);
        emit SoldStableCoins(msg.sender, receiver, amountSC, amountBC);
    }
```

Listing 3.3: Djed.sol

Recommendation Revisit the above mentioned routines to add proper validations for the buy/sell of the `stablecoin/reservecoin` to better protect users funds.

Status The issue has been fixed by this commit: `b452f0f2`.

4 Conclusion

In this audit, we have analyzed the `Djed` protocol design and implementation. `Djed` is an autonomous crypto-backed stable coin protocol which maintains a reserve of the EVM-compatible blockchain's independent native currency to back the stable coin in circulation. `Djed` issues two ERC20-compliant coins: the `stablecoin` and the `reservecoin`, which can be bought/sold directly with `Djed`. During the audit, we notice that the current code base is well organized and those identified issues are promptly fixed.

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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

[5] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.