SMART CONTRACT AUDIT REPORT

for

TChoke

Prepared By: Xiaomi Huang

PeckShield

February 14, 2024

Document Properties

Client	TChoke
Title	Smart Contract Audit Report
Target	TChoke
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Final

Version Info

Version	Date	Author(s)	Description
1.0	February 14, 2024	Xuxian Jiang	Final Release
1.0-rc1	February 13, 2024	Xuxian Jiang	Release Candidate #1

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 TChoke 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 TChoke

Artichoke is a liquidity provision protocol housed on the Arbitrum One blockchain. It aims to enhance on-chain capital efficiency by allowing users to provide single-sided liquidity through its infrastructure tooling that enables the provision of one-sided liquidity to be added to any token. Artichoke benefits protocols and investors by reducing the need for token incentives to develop robust liquidity pools and mitigating impermanent loss for LPs (Liquidity Providers). For example, it allows to stake Camelot's spNFT positions to mint tCHOKE that represents the minted fixed percentage of USDC according to the position size. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of TChoke

Item	Description
Name	TChoke
Website	https://articho.ke/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 14, 2024

In the following, we show the deployment address of the audited TChoke contract:

- TChoke: https://arbiscan.io/token/0x110975fdd26f397eab71233c560d34ba01792853

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

- https://github.com/0xCCLVI/tchoke.git (be29d67)

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 the 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

To evaluate the risk, we go through a checklist of 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 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 implementation of the `TChoke` protocol. 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 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 TChoke Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor Logic in TChoke	Coding Practices	Resolved
PVE-002	Low	Inconsistent Liquidity Source Enforcement in TChoke	Coding Practices	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Mitigated

Besides 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 Improved Constructor Logic in TChoke

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

Description

To facilitate possible future upgrade, the `TChoke` constract is instantiated as a proxy with actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers();`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
    function initialize(uint256 initialSupply) public initializer {
        __ReentrancyGuard_init();
        __AccessControl_init();
        __ERC20_init("tChoke", "tChoke");
        __ERC20Burnable_init();

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(LIQUIDITY_MANAGER_ROLE, msg.sender);
        _grantRole(DEBT_MANAGER_ROLE, msg.sender);

        _mint(msg.sender, initialSupply);
    }
```

Listing 3.1: `TChoke::initialize()`

Recommendation Improve the above-mentioned constructor routine in `TChoke`.

Status This issue has been fixed by the following commit: `10ac03b`.

3.2 Inconsistent Liquidity Source Enforcement in TChoke

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

Description

The `TChoke` contract allows to add, remove, or configure the supported liquidity sources. In the process of examining the liquidity source management, we notice a minor inconsistency when validating the given liquidity source.

Specifically, we show below the implementation of related routines, i.e., `addLiquiditySource()` and `deposit()`. The former routine adds a new liquidity source and the latter allows for depositing into the given liquidity source. However, it comes to our attention that the liquidity source can be added without requiring `liquiditySource != address(0)` and the liquidity may not be deposited if `liquiditySource == address(0)`. A better suggestion is to revise the liquidity source so that the given `liquiditySource` should not be `address(0)`.

```
    function addLiquiditySource(
        address liquiditySource,
        address handler
    ) external onlyRole(LIQUIDITY_MANAGER_ROLE) {
        if (
            handler == address(0)
            liquiditySources[liquiditySource].handler != address(0)
            liquiditySources[liquiditySource].debt != 0
        ) revert TChokeInvalidHandler();

        liquiditySources[liquiditySource] = LiquiditySourceHandler(
            0,
            0,
            handler,
            false
        );

        emit AddLiquiditySource(liquiditySource, handler);
    }
```

Listing 3.2: `TChoke::addLiquiditySource()`

```
    function deposit(
        address liquiditySource,
        uint256 positionID
    ) external nonReentrant {
        if (totalDebt > totalDebtCeiling) revert TChokeDebtCeilingExceeded();

        LiquiditySourceHandler storage handler = liquiditySources[
            liquiditySource
        ];

        if (
            liquiditySource == address(0)
            handler.handler == address(0)
            positionID == 0
            handler.debt > handler.debtCeiling
        ) revert TChokeInvalidLiquidityPosition(liquiditySource);
        ...
    }
```

Listing 3.3: `TChoke::deposit()`

Recommendation Revise the above functions to ensure the liquidity source validation is consistent.

Status This issue has been fixed by the following commit: `be29d67`.

3.3 Trust Issue of Admin Keys

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

Description

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

```
    function addLiquiditySource(
        address liquiditySource,
        address handler
```

```
    ) external onlyRole(LIQUIDITY_MANAGER_ROLE) {
        if (
            handler == address(0)
            liquiditySources[liquiditySource].handler != address(0)
            liquiditySources[liquiditySource].debt != 0
        ) revert TChokeInvalidHandler();

        liquiditySources[liquiditySource] = LiquiditySourceHandler(
            0,
            0,
            handler,
            false
        );

        emit AddLiquiditySource(liquiditySource, handler);
    }

    /**
     * @dev To be executed by any user with `LIQUIDITY_MANAGER` permission.
     * Removing any liquiditySource must be consistent with `totalDebt` and `
         totalDebtCeiling`
     * In order to ensure liquidity parameters invariant, `handler.debt` and `handler.
         debtCeiling` MUST be zero.
     * @param liquiditySource Any address representing underlying liquidity previously
         added using {addLiquiditySource}.
     *
     * Emits a {RemoveLiquiditySource} event.
     *
     */

    function removeLiquiditySource(
        address liquiditySource
    ) external onlyRole(LIQUIDITY_MANAGER_ROLE) {
        if (
            liquiditySources[liquiditySource].handler == address(0)
            liquiditySources[liquiditySource].debt != 0
            liquiditySources[liquiditySource].debtCeiling != 0
        ) revert TChokeInvalidHandler();

        liquiditySources[liquiditySource] = LiquiditySourceHandler(
            0,
            0,
            address(0),
            false
        );

        emit RemoveLiquiditySource(liquiditySource);
    }

    /**
     * @dev To be executed by any user with `DEBT_MANAGER` permission.
     * Single entrypoint for modifying totalDebtCeiling.
```

```
     * In order to ensure liquidity parameters invariant, `handler.debt` and `handler.
       debtCeiling` MUST be zero.
     * @param liquiditySource Any address representing underlying liquidity previously
       added using {addLiquiditySource}.
     * @param _debtCeiling New `debtCeiling` for specified `liquiditySource`. Previous `
       handler.debtCeiling` will be subtracted from `totalDebtCeiling` first.
     * @param _paused Will pause liquiditySource and disable depositing for that
       specific source.
     *
     * Emits a {SetDebtCeiling} event.
     */

    function setDebtCeiling(
        address liquiditySource,
        uint256 _debtCeiling,
        bool _paused
    ) external onlyRole(DEBT_MANAGER_ROLE) {
        if (liquiditySources[liquiditySource].handler == address(0))
            revert TChokeInvalidHandler();

        totalDebtCeiling =
            totalDebtCeiling -
            liquiditySources[liquiditySource].debtCeiling;

        liquiditySources[liquiditySource].debtCeiling = _debtCeiling;

        totalDebtCeiling = totalDebtCeiling + _debtCeiling;

        liquiditySources[liquiditySource].paused = _paused;

        emit SetDebtCeiling(liquiditySource, _debtCeiling, _paused);
    }
```

Listing 3.4: Example Privileged Operations in `TChoke` Contract

Apparently, if the privileged account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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 mitigated with the use of `multisig` to manage the admin key.

4 Conclusion

In this audit, we have analyzed the design document and related smart contract source code of the TChoke protocol, which is a liquidity provision protocol housed on the Arbitrum One blockchain. It aims to enhance on-chain capital efficiency by allowing users to provide single-sided liquidity through its infrastructure tooling that enables the provision of one-sided liquidity to be added to any token. Artichoke benefits protocols and investors by reducing the need for token incentives to develop robust liquidity pools and mitigating impermanent loss for LPs (Liquidity Providers). For example, it allows to stake Camelot's spNFT positions to mint tCHOKE that represents the minted fixed percentage of USDC according to the position size. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that Solidity-based 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.