SMART CONTRACT AUDIT REPORT

for

ZKT

Prepared By: Yiqun Chen

PeckShield

August 9, 2021

Document Properties

Client	zkTube
Title	Smart Contract Audit Report
Target	ZKT
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 9, 2021	Xiaotao Wu	Final Release
1.0-rc	August 7, 2021	Xiaotao Wu	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the ZKT smart contracts, 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 ZKT

As the native token of the zkTube protocol, ZKT represents the rights of the holder and also has other practical uses. The usage scenarios of ZKT include certificate of community participation in governance, transaction fee and fuel in the zkTube network, mining, cryptocurrency assets, and circulation in DeFi and NFT.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of ZKT

Item	Description
Name	zkTube
Website	https://zktube.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 9, 2021

In the following, we show the MD5 hash value of the related compressed file with the contracts for audit:

- MD5 (ZKTContracts_audit.zip) = 26a3b924a12783f0f8f9bc2043b3fa88

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 accordingly classified into four categories, 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. 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 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 design and implementation of the ZKT smart contracts. 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	0	
Medium	1	■
Low	1	■
Informational	1	■
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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Sanity Checks For System Parameters	Coding Practices	Confirmed
PVE-002	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Medium	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 Improved Sanity Checks For System Parameters

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

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The ZKT protocol is no exception. Specifically, if we examine the `ZKTDeposit` contract, it has defined a system-wide risk parameter: `lockTime`. In the following, we show the corresponding routine that allows for its changes.

```
    function updateLockTime(uint lockTime_) external onlyOwner {
        lockTime = lockTime_;
        emit UpdateLockTime(msg.sender, lockTime_);
    }
```

Listing 3.1: `ZKTDeposit::updateLockTime()`

This parameter defines the lock time when `zkTube` users deposit their `ZKT` tokens into the `ZKTDeposit` contract and need to exercise extra care when configuring or updating it. Our analysis shows the update logic on this parameter can be improved by applying more rigorous sanity check. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `lockTime` may set a huge lock time, resulting in the `zkTube` users' assets permanently locked in the `ZKTDeposit` contract.

Recommendation Validate any change regarding this system-wide parameter to ensure it fall in an appropriate range.

Status The issue has been confirmed.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ZKTWhiteList`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In ZKT, the `ZKTWhiteList` contract is designed for `zkTube` users to change their `ZKTR` tokens to `ZKT` tokens. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., `ZKTR`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the pool. 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 contract. In the following, we show the `deposit()` routine that is used to deposit `ZKTR` tokens to the `ZKTWhiteList` contract.

```
    function deposit(uint amount) external whenNotPaused lock {
        require(amount > 0, "ZKTWhiteList: amount is zero");
        zktrToken.safeTransferFrom(msg.sender, address(this), amount);
        _addDeposit(msg.sender, amount);
        deposits[msg.sender] = deposits[msg.sender] + amount;
        totalDeposits = totalDeposits + amount;
        emit Deposit(msg.sender, amount);
    }

    function withdraw(uint amount) external lock {
        require(amount > 0, "ZKTWhiteList: amount is zero");
        require(amount <= _available(msg.sender, getCurrentTime() / ONE_DAY), "
            ZKTWhiteList: available is not enough");
        // update
        _addReleased(msg.sender, amount);
        withdrawals[msg.sender] = withdrawals[msg.sender] + amount;
        totalWithdrawals = totalWithdrawals + amount;
        // transfer
        zktToken.safeTransfer(msg.sender, amount);
        emit Withdraw(msg.sender, amount);
    }
```

Listing 3.2: `ZKTWhiteList::deposit()/withdraw()`

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 a 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 expecting the amount parameter in transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the ZKTWhiteList contract before and after the 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.

We emphasize that the current implementation of ZKTWhiteList is safe as it accepts none-deflationary tokens, i.e., ZKTR for deposits. However, the current code implementation is generic in supporting various tokens and there is a need to highlight the possible pitfall from the audit perspective.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the safeTransferFrom() 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 This issue has been confirmed.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the ZKT contracts, there are two special administrative accounts, i.e., _owner and admin. These two privileged accounts play critical roles in governing and regulating the entire operation and maintenance. We examine closely the ZKT contracts and identify below trust issues on these two privileged accounts.

Firstly, we note that the pause() and unpause() functions allow for the _owner to set the ZKTWhiteList contract state to be paused or unpaused. The zkTube users can only deposit their ZKTR tokens when ZKTWhiteList contract is in unpaused state.

```
    function pause() public onlyOwner {
        _pause();
    }

    function unpause() public onlyOwner {
        _unpause();
    }
```

Listing 3.3: `ZKTWhiteList::pause()/unpause()`

Secondly, we note that the `updateLockedPosition()` function of `ZKTVesting` contract allows for the `_owner` to update the `lockedPosition`. This state variable determines the number of `ZKT` token rewards that the `zkTube` users can claim every day.

```
    function updateLockedPosition(uint lockedPosition_) external onlyOwner {
        require(lockedPosition_ <= 100, "ZKTVesting: lockedPosition too big");
        lockedPosition = lockedPosition_;
        emit UpdateLockedPosition(msg.sender, lockedPosition_);
    }
```

Listing 3.4: `ZKTVesting::updateLockedPosition()`

Thirdly, we note that the `updateLockTime()` function of `ZKTDeposit` contract allows for the `_owner` to update the `lockTime`. This state variable determines how long the users‘ `ZKT` token will be locked in the `ZKTDeposit` contract.

```
    function updateLockTime(uint lockTime_) external onlyOwner {
        lockTime = lockTime_;
        emit UpdateLockTime(msg.sender, lockTime_);
    }
```

Listing 3.5: `ZKTDeposit::updateLockTime()`

Lastly, we note that the `admin` account has the power to update the logic implementations for contracts `ZKTWhiteList`, `ZKTVesting`, and `ZKTDeposit`.

```
import "@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";

contract ZKTWhiteListUpgradeableProxy is TransparentUpgradeableProxy {
    constructor(address logic, address admin, bytes memory data)
        TransparentUpgradeableProxy(logic, admin, data) {

    }
}
```

Listing 3.6: `ZKTWhiteListUpgradeableProxy.sol`

```
import "@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";

contract ZKTVestingUpgradeableProxy is TransparentUpgradeableProxy {
    constructor(address logic, address admin, bytes memory data)
        TransparentUpgradeableProxy(logic, admin, data) {
```

```

    }
}
```

Listing 3.7: ZKTVestingUpgradeableProxy.sol

```
import "@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";

contract ZKTDepositUpgradeableProxy is TransparentUpgradeableProxy {
    constructor(address logic, address admin, bytes memory data)
        TransparentUpgradeableProxy(logic, admin, data) {

    }
}
```

Listing 3.8: ZKTDepositUpgradeableProxy.sol

We understand the need of the privileged functions for contract operation, but at the same time the extra power to the `_owner` and `admin` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to `_owner` and `admin` explicit to `zkTube` users.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the ZKT design and implementation. As the value carrier of zkTube, ZKT supports staking, CPU mining and import wallet mining, and the income is ZKT. The current code base is well organized and those identified issues are promptly confirmed and resolved.

Meanwhile, 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.