SMART CONTRACT AUDIT REPORT

for

TARS Protocol - Claimer

Prepared By: Xiaomi Huang

PeckShield

July 20, 2022

Document Properties

Client	TARS
Title	Smart Contract Audit Report
Target	Claimer
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 20, 2022	Xuxian Jiang	Final Release
1.0-rc	July 18, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Claimer` 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 Claimer

`Claimer` is a dApp powered by `Dynamic Penalty Function (DPF)` and allows project teams/ token purchasers to lock tokens in a non-custodial, time-released smart contract vault, and whitelisted users will be able to claim their specified tokens according to `Token Vesting Schedules`. As a once-and-for-all `Token Vesting Schedule` with few clicks, `Claimer` will allow all investors to claim specified tokens fairly/timely with no more complaints. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The `Stader` Protocol

Item	Description
Issuer	TARS
Website	http://docs.tars.pro
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 20, 2022

In the following, we show the two contract files for audit and the MD5/SHA checksum value of each:

- Name: Claimer.sol (MD5: 70437777e7be8e06a8c438d107529001)

- Name: FactoryIDO.sol (MD5: 3087b6ccd2027b60eed23f8117ef7002)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

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) [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 design and implementation of the `Claimer` 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	2	■ ■
Informational	0	
Total	3	

We have so far identified a list of potential issues.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 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Claimer Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possibly Repeated Refunds in FactoryIDO	Business Logic	Resolved
PVE-002	Low	Integer Overflow Avoidance With SafeMath Enforcement	Numeric Errors	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Possibly Repeated Refunds in FactoryIDO

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FactoryIDO`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `FactoryIDO` contract has a function that allows to request the refund of full `totalQuota` back to the claimer owner. While analyzing the related refund logic, we notice its current implementation may allow for repeated refunds.

To elaborate, we show below this function `refundClaimer()`. It has a rather straightforward logic in paying the associated refund price, and then invoking the refund logic of the claimer contract. It comes to our attention that this function does not validate the requested refund has occured before or not. As a result, a malicious actor may repeatedly invoke a claimer's refund logic. (Note each refund requires an associated refund price as the payment.)

```
function refundClaimer(address _claimerAddr) public payable isClaimerContract(
    _claimerAddr) isClaimerOwner(_claimerAddr) {
  require(msg.value >= globalParams.refundPrice, "Err_F_04");
  IClaimer(_claimerAddr).refund();
  claimerContractInfo[_claimerAddr].isRefund = true;
  if (globalParams.refundPrice > 0) {
    payee.transfer(globalParams.refundPrice);
  }

}
```

Listing 3.1: `FactoryIDO::refundClaimer()`

Recommendation Only allow for the refund if the claimer has not refunded before.

Status This issue has been fixed by validating that the given `_claimerAddr` has not been refunded yet.

3.2 Integer Overflow Avoidance With SafeMath Enforcement

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Claimer`
- Category: Numeric Errors [6]
- CWE subcategory: CWE-190 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. In this section, we examine one possible case where the `SafeMath` calculation is not applied.

In particular, we use below the `Claimer::initialize()` function. This routine is used to properly configure the claim rates for different participants. However, we notice the sum of the provided claim rates is validated to be `10000` (line 165). But the addition of two claim rates is not performed with `SafeMath`. To avoid unnecessary overflows, there is a need to use `SafeMath` for the arithmetic operations.

```
113     function initialize(
114         bool _isERC20,
115         address[] calldata _addressAddr,
116         string[] calldata _requireStringArr,
117         string[] calldata _optionalStringArr,
118         uint256[] calldata _dates,
119         uint256[] calldata _claimRates
120     ) external {
121         require(assemblyLineAddr == msg.sender, "Err_C_02");

123         isERC20 = _isERC20;
124         tokenAddr = _addressAddr[0]; //
125         claimerOwner = _addressAddr[1];
126         superAdmin = _addressAddr[2];
127         factoryAddr = _addressAddr[3];

129         require(tokenAddr != address(0), "Token can not be 0x0!");
130         require(claimerOwner != address(0), "Claim owner can not be 0x0!");
131         require(superAdmin != address(0), "Super admin can not be 0x0!");
132         require(_claimRates.length > 0 && _claimRates.length == _dates.length, "Err_C_04
                ");
133         require(_dates[0] > block.timestamp, "Err_C_05");
```

```
        claimerReqInfo.claimerName = _requireStringArr[0];
        claimerReqInfo.tokenSymbol = _requireStringArr[1];
        claimerReqInfo.tokenLogo = _requireStringArr[2];
        claimerReqInfo.decimals = IERC20Metadata(tokenAddr).decimals();

        claimerOptInfo.homePage = _optionalStringArr[0];
        claimerOptInfo.twitter = _optionalStringArr[1];
        claimerOptInfo.discord = _optionalStringArr[2];
        claimerOptInfo.tg = _optionalStringArr[3];
        claimerOptInfo.auditReport = _optionalStringArr[4];
        claimerOptInfo.CMC = _optionalStringArr[5];
        claimerOptInfo.cg = _optionalStringArr[6];

        beginRuleTime = _dates[0];

        addAuth(superAdmin);
        addAuth(claimerOwner);

        uint256 count = 0;
        for (uint256 i = 0; i < _claimRates.length; i++) {
            if (i > 0) {
                require(_dates[i] > claimRuleInfoArray[i - 1].date, "Err_C_05");
            }

            count += _claimRates[i];
            ClaimRuleInfo memory obj = ClaimRuleInfo({date: _dates[i], claimRate: _claimRates[i]});
            claimRuleInfoArray.push(obj);

            initTimeForClaimRule.push(_dates[i]);
        }
        require(count == 10000, "Err_C_07");
    }
```

Listing 3.2: Claimer:: initialize ()

Recommendation Revise the above calculations to make use of `SafeMath` against unexpected arithmetic overflows or underflows.

Status The issue has been fixed with the suggested use of `SafeMath`.

3.3 Trust Issue of Admin Keys

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

Description

In the `Claimer` contracts, there is a privileged manager account `admin` that plays a critical role in governing and regulating the system-wide operations (e.g., authorize other accounts as well as transfer funds out of `Claimer`). Our analysis shows that the privileged account needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the privileged account. Specifically, the privileged functions in the `FactoryIDO` contract allow for the withdrawal of all funds from the `Claimer` contract.

```
function adminSafeTransfer(
  address _token,
  address _to,
  uint256 _amount
) public onlySuperAdmin(msg.sender) {
  _upgradeSafeTransfer(_token, _to, _amount);
}

function updateAdmin(address _admin) public onlySuperAdmin(msg.sender) {
  removeAuth(admin);
  admin = _admin;
  addAuth(admin);
}

function updatePayee(address _payee) public onlyOperator {
  require(_payee != address(0), "Payee can not be 0x0!");
  payee = address(uint160(_payee));
}
```

Listing 3.3: Example Privileged Operations in `FactoryIDO`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account 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 protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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 The issue has been mitigated by the team by removing the privileged function `adminSafeTransfer()`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Claimer` smart contracts, which are a dApp powered by `Dynamic Penalty Function (DPF)` and allow project teams/ token purchasers to lock tokens in a non-custodial, time-released smart contract vault. Whitelisted users will be able to claim their specified tokens according to `Token Vesting Schedules`. As a once-and-for-all `Token Vesting Schedule` with few clicks, `Claimer` will allow all investors to claim specified tokens fairly/timely with no more complaints. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-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-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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.