SMART CONTRACT AUDIT REPORT

for

ARPA Staking v0.1

Prepared By: Xiaomi Huang

PeckShield

June 1, 2023

Document Properties

Client	ARPA
Title	Smart Contract Audit Report
Target	ARPA Staking v0.1
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Jing Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 1, 2023	Jing Wang	Final Release
1.0-rc	May 12, 2023	Jing Wang	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 `ARPA Staking` protocol design document and related smart contract source code, 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 branch of `ARPA Staking` protocol 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 ARPA Staking

`ARPA Staking` is a staking protocol similar to the `Chainlink` staking, managing the `ERC20` assets staked by the node operators as well as token holders. These token holders will be able to stake their tokens into a certain pool of the operator and earn rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ARPA Staking v0.1

Item	Description
Name	ARPA
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 1, 2023

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

- https://github.com/ARPA-Network/Staking-v0.1 (37c99a5)

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

- https://github.com/ARPA-Network/Staking-v0.1 (aea24e44)

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 `ARPA Staking` protocol 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	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 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 3 low-severity vulnerabilities.

Table 2.1: Key ARPA Staking v0.1 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Time and State	Fixed
PVE-002	Low	Potential Reentrancy Risk in Staking	Time and State	Fixed
PVE-003	Low	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 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

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

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below.

```
/**
* @dev transfer token for a specified address
* @param _to The address to transfer to.
* @param _value The amount to be transferred.
*/
function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    uint sendAmount = _value.sub(fee);
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(msg.sender, owner, fee);
    }
```

```
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer(address to, uint tokens)virtual public returns (bool success)`. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `unstake()` routine in the `Staking` contract. If `USDT` is given as `i_ARPA`, the unsafe version of `IERC20(token).transfer(to, amount)` (line 344) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
334     /// @inheritdoc IStaking
335     function unstake(uint256 amount) external override(IStaking) whenNotPaused {
336         // Round down unstake amount to avoid cumulative rounding errors.
337         uint256 remainder = amount % RewardLib.REWARD_PRECISION;
338         if (remainder > 0) {
339             amount -= remainder;
340         }

342         (uint256 baseReward, uint256 delegationReward) = _exit(msg.sender, amount, false
            );

344         i_ARPA.transfer(msg.sender, baseReward + delegationReward);

346         emit Unstaked(msg.sender, amount, baseReward, delegationReward);
347     }
```

Listing 3.2: `Staking::unstake()`

Note that other routines in the `Staking` contracts share the same issue.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()/transferFrom()`.

Status This issue has been fixed in the commit: 1f80a38.

3.2 Potential Reentrancy Risk in Staking

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the `Uniswap/Lendf.Me` hack [10].

We notice there is an occasions where the `checks-effects-interactions` principle is violated. Using the `Staking` contract as an example, the `stake()` function (see the code snippet below) is provided to enter staking. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 324) starts before effecting the update on internal states (lines 327 and 329), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same function.

```
312     /// @inheritdoc IStaking
313     function stake(uint256 amount) external override(IStaking) whenNotPaused {
314         if (amount < RewardLib.REWARD_PRECISION) {
315             revert StakingPoolLib.InsufficientStakeAmount(RewardLib.REWARD_PRECISION);
316         }

318         // Round down input amount to avoid cumulative rounding errors.
319         uint256 remainder = amount % RewardLib.REWARD_PRECISION;
320         if (remainder > 0) {
321             amount -= remainder;
322         }

324         i_ARPA.transferFrom(msg.sender, address(this), amount);

326         if (s_pool._isOperator(msg.sender)) {
327             _stakeAsOperator(msg.sender, amount);
328         } else {
329             _stakeAsCommunityStaker(msg.sender, amount);
```

```
        }
    }
```

Listing 3.3: `Staking::stake()`

Recommendation Apply the non-reentrancy protection in all above-mentioned routines.

Status This issue has been fixed in the commit: 34a415f.

3.3 Trust Issue of Admin Keys

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

Description

In the `ARPA Staking` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure protocol parameters). 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 `owner` account and its related privileged accesses in current contract.

To elaborate, we show below the related function. The `setController()` routine supports the configuration of `s_controller` value, which is a key parameter to lock/unlock tokens.

```
    function setController(address controller) external override(IStakingOwner)
        onlyOwner {
        if (controller == address(0)) revert InvalidZeroAddress();
        s_controller = controller;

        emit ControllerSet(controller);
    }

    function lock(address staker, uint256 amount) external override(INodeStaking)
        onlyController {
        StakingPoolLib.Staker storage stakerAccount = s_pool.stakers[staker];
        if (!stakerAccount.isOperator) {
            revert StakingPoolLib.OperatorDoesNotExist(staker);
        }
        if (stakerAccount.stakedAmount < amount) {
            revert StakingPoolLib.InsufficientStakeAmount(amount);
        }
        stakerAccount.lockedStakeAmount += amount._toUint96();
        emit Locked(staker, amount);
```

```
    }
```

Listing 3.4: `Staking::setController()and lock()`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` account is a plain EOA account. 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.

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 The issue has been confirmed by the team. The team clarifies they will transfer owner to `DAO` once the contract is online.

4 Conclusion

In this audit, we have analyzed the `ARPA Staking` protocol design and implementation. The `ARPA Staking` protocol is a staking protocol which manages the `ERC20` assets staked by the node operators as well as token holders. 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-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-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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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