



SMART CONTRACT AUDIT REPORT

for

DAO Staking



Prepared By: Yiqun Chen

PeckShield
November 15, 2021

Document Properties

Client	DaoMaker
Title	Smart Contract Audit Report
Target	DAO Staking
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 15, 2021	Xuxian Jiang	Final Release
1.0-rc	November 12, 2021	Xuxian Jiang	Release Candidate

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	4
1.1	About DAO Maker	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Inconsistency Between Document and Implementation	11
3.2	Suggested Adherence Of Checks-Effects-Interactions Pattern	13
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the **DAO Staking** 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 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 DAO Maker

DAO Maker aims to develop the go-to platform for retail venture investing in equity and tokens. Providing low-risk participation frameworks is essential to reach global retail in venture capital, as most retail investors cannot afford to risk large portions of their money. By providing an opportunity to everyday people to safely grow their own capital, DAO Maker improves the quality of millions of lives while simultaneously enables a new funding source to innovation worldwide. The audited DAO Staking provides the opportunities for the protocol users to stake and get rewards.

The basic information of DAO Staking is as follows:

Table 1.1: Basic Information of DAO Staking

Item	Description
Name	DaoMaker
Website	https://daomaker.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 15, 2021

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

- <https://github.com/daomaker/dao-staking.git> (e216282)

1.2 About PeckShield

PeckShield Inc. [7] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- 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 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

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

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) [5], 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 DAO Staking 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	1	
Informational	1	
Total	2	

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 low-severity vulnerability and 1 informational recommendation.

Table 2.1: Key DAO Staking Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Inconsistency Between Document and Implementation	Coding Practices	Confirmed
PVE-002	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	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 Inconsistency Between Document and Implementation

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

As an example, the protocol provides bonus shares for a new stake and the bonus shares are measured based on the staking duration and the staking amount. And a helper routine `stakeStartBonusShares()` is provided for this purpose. According to the comments (lines 384-388), the maximum days for the longer pays better (LPB) is computed as `LPB_MAX_DAYS = 3640`. However, it is defined as a constant of 1080 (line 78).

```
209     function stakeStartBonusShares(uint256 newStakedAmount, uint256 newStakedDays)
210     public
211     pure
212     returns (uint256 bonusShares)
213     {
214         /*
215             LONGER PAYS BETTER:
216
217             If longer than 1 day stake is committed to, each extra day
218             gives bonus shares of approximately 0.0548%, which is approximately 20%
219             extra per year of increased stake length committed to, but capped to a
220             maximum of 200% extra.
221
222             extraDays          =   stakedDays - 1
223
224             longerBonus%      = (extraDays / 364) * 20%
```

```

225         = (extraDays / 364) / 5
226         = extraDays / 1820
227         = extraDays / LPB
228
229     extraDays      = longerBonus% * 1820
230     extraDaysMax   = longerBonusMax% * 1820
231                   = 200% * 1820
232                   = 3640
233                   = LPB_MAX_DAYS
234
235     BIGGER PAYS BETTER:
236
237     Bonus percentage scaled 0% to 10% for the first 150M of stake.
238
239     biggerBonus%   = (cappedStake / BPB_MAX) * 10%
240                   = (cappedStake / BPB_MAX) / 10
241                   = cappedStake / (BPB_MAX * 10)
242                   = cappedStake / BPB
243
244     COMBINED:
245
246     combinedBonus% =          longerBonus%  + biggerBonus%
247
248                   cappedExtraDays    cappedStake
249                   ----- + -----
250                   LPB                BPB
251
252                   cappedExtraDays * BPB    cappedStake * LPB
253                   ----- + -----
254                   LPB * BPB                LPB * BPB
255
256                   cappedExtraDays * BPB + cappedStake * LPB
257                   -----
258                   LPB * BPB
259
260     bonusShares    = stake * combinedBonus%
261                   = stake * (cappedExtraDays * BPB + cappedStake * LPB) / (
262                               LPB * BPB)
263
264     /*
265     /* Must be more than 1 day for Longer-Pays-Better */
266     if (newStakedDays > 1) {
267         cappedExtraDays = newStakedDays <= LPB_MAX_DAYS ? newStakedDays - 1 :
268                               LPB_MAX_DAYS;
269     }
270
271     uint256 cappedStakedAmount = newStakedAmount >= BPB_FROM_AMOUNT ?
272         newStakedAmount - BPB_FROM_AMOUNT : 0;
273     if (cappedStakedAmount > BPB_MAX) {
274         cappedStakedAmount = BPB_MAX;
275     }

```

```

274
275     bonusShares = cappedExtraDays * BPB + cappedStakedAmount * LPB;
276     bonusShares = newStakedAmount * bonusShares / (LPB * BPB);
277
278     return bonusShares;
279 }

```

Listing 3.1: Staking::stakeStartBonusShares()

Similarly, the BPB is defined as $BPB = BPB_MAX * 100 / BPB_BONUS_PERCENT = BPB_MAX * 2$ (line 83), while the associated comment indicates $BPB = BPB_MAX * 10$ (lines 396-397).

```

74  /* Stake shares Longer Pays Better bonus constants used by _stakeStartBonusShares()
    */
75  uint256 private constant LPB_BONUS_PERCENT = 600;
76  uint256 private constant LPB_BONUS_MAX_PERCENT = 1800;
77  uint256 internal constant LPB = 364 * 100 / LPB_BONUS_PERCENT;
78  uint256 internal constant LPB_MAX_DAYS = LPB * LPB_BONUS_MAX_PERCENT / 100;
79
80  /* Stake shares Bigger Pays Better bonus constants used by _stakeStartBonusShares()
    */
81  uint256 private constant BPB_BONUS_PERCENT = 50;
82  uint256 internal constant BPB_MAX = 1e6 * 10 ** TOKEN_DECIMALS;
83  uint256 internal constant BPB = BPB_MAX * 100 / BPB_BONUS_PERCENT;
84  uint256 internal constant BPB_FROM_AMOUNT = 50000 * 10 ** TOKEN_DECIMALS;

```

Listing 3.2: GlobalsAndUtility.sol

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been confirmed.

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [4]
- CWE subcategory: CWE-663 [2]

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 [9] exploit, and the recent Uniswap/Lendf.Me hack [8].

We notice there is an occasion where the checks-effects-interactions principle is violated. Using the Staking as an example, the `stakeStart()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 52) starts before effecting the update on internal states (lines 54), 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 same entry function.

```

35     function stakeStart(uint256 newStakedAmount, uint256 newStakedDays)
36         external
37     {
38         GlobalsCache memory g;
39         _globalsLoad(g);
40
41         /* Enforce the minimum stake time */
42         require(newStakedDays >= MIN_STAKE_DAYS, "STAKING: newStakedDays lower than
           minimum");
43         /* Enforce the maximum stake time */
44         require(newStakedDays <= MAX_STAKE_DAYS, "STAKING: newStakedDays higher than
           maximum");
45
46         /* Check if log data needs to be updated */
47         _dailyDataUpdateAuto(g);
48
49         _stakeStart(g, newStakedAmount, newStakedDays);
50
51         /* Remove staked amount from balance of staker */
52         stakingToken.safeTransferFrom(msg.sender, address(this), newStakedAmount);
53
54         _globalsSync(g);
55     }

```

Listing 3.3: Staking::stakeStart()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy. However, it is important to take precautions in making use of `nonReentrant` to block possible re-entrancy. Note similar issues exist in other functions, including `Staking::stakeEnd()` and the adherence of checks-effects-interactions best practice is recommended.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible re-entrancy.

Status The issue has been confirmed.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the DAO Staking protocol. The system presents a unique offering as a decentralized protocol to allow for protocol users to stake and get rewards. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [3] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [4] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [5] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [8] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [9] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.