SMART CONTRACT AUDIT REPORT

for

Tidal Governance

Prepared By: Shuxiao Wang

PeckShield

June 3, 2021

Document Properties

Client	Tidal
Title	Smart Contract Audit Report
Target	Tidal Governance
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	June 3, 2021	Xuxian Jiang	Release Candidate #1
0.2	May 31, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 25, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction **4**

1.1 About Tidal 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 6

2 Findings **10**

2.1 Summary 10

2.2 Key Findings 11

3 Detailed Results **12**

3.1 Proper Membership Removal in removeMember() 12

3.2 Suggested Reentrancy Prevention in CommitteeAlpha And Staking 13

3.3 Potential Bypass Of Staking Withdraw Waiting 14

3.4 Block Time-Related votingPeriod Adjustment 15

4 Conclusion **17**

References **18**

1 Introduction

Given the opportunity to review the **Tidal Governance** 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 Tidal

`Tidal` is a decentralized discretionary mutual cover protocol that offers the DeFi community the ability to hedge against the failure of any DeFi protocol or asset. By directly leveraging up the reserve to cover multiple protocols at the same time, the enhanced capital efficiency attracts liquidity providers (LPs) while a competitive insurance premium attracts buyers. `Tidal` pools capital from sellers to offer covers to purchasers. This allows for higher capital efficiency as the same reserve backs more covers than can be individually paid out and also eliminates a peer matching process resulting from double coincidence of wants. This audit covers its `governance` subsystem.

The basic information of Tidal Governance is as follows:

Table 1.1: Basic Information of Tidal Governance

Item	Description
Issuer	Tidal
Website	https://tidal.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 3, 2021

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

this audit. Note that Tidal Governance assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/TidalFinance/tidal-contracts.git (1f5ac0b)

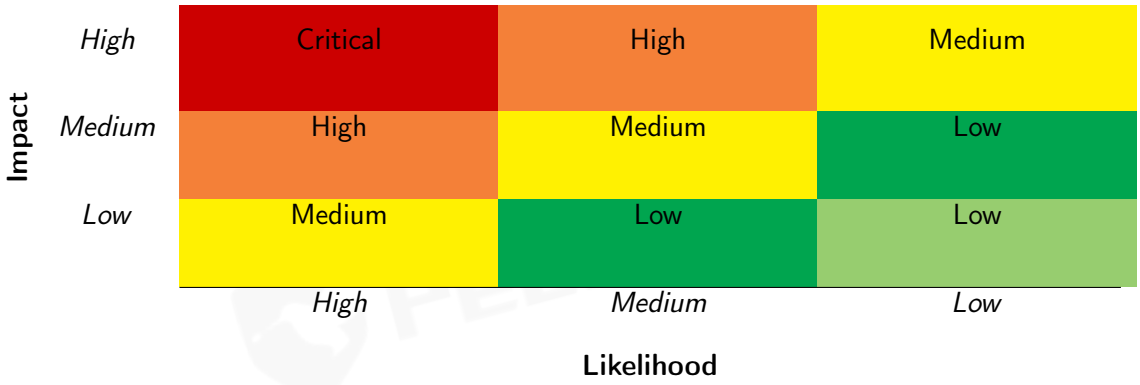
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/TidalFinance/tidal-contracts.git (058f1ec)

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



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.

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

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Tidal Governance 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	1	■
Low	3	■■■
Informational	0	
Total	4	

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, and 3 low-severity vulnerabilities.

Table 2.1: Key Tidal Governance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Membership Removal in removeMember()	Business Logic	Fixed
PVE-002	Low	Suggested Reentrancy Prevention in CommitteeAlpha And Staking	Time and State	Fixed
PVE-003	Low	Potential Bypass Of Staking Withdraw Waiting	Business Logic	Fixed
PVE-004	Low	Block Time-Related votingPeriod Adjustment	Coding Practices	Fixed

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 Proper Membership Removal in removeMember()

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

Description

The `Tidal` protocol has a `CommitteeAlpha` contract that is designed to allow a committee to be organized. With that, it provides two membership-management functions: `addMember()` and `removeMember()`. While examining these two functions, we notice the `removeMember()` function needs to be improved.

To elaborate, we show below the `removeMember()` function. It implements a rather straightforward logic in locating and removing the given member from the committee. However, it comes to our attention that while it properly removes the given member from the internal array of `members`, it does not timely update another related `memberIndexPlusOne` array.

```
79      function removeMember(address who_) external onlyOwner {
80          uint256 indexPlusOne = memberIndexPlusOne[who_];
81          require(indexPlusOne > 0, "Invalid address");
82          require(indexPlusOne <= members.length, "Out of range");
83          if (indexPlusOne < members.length) {
84              members[indexPlusOne.sub(1)] = members[members.length.sub(1)];
85              memberIndexPlusOne[members[indexPlusOne.sub(1)]] = indexPlusOne;
86          }
87
88          members.pop();
89      }
```

Listing 3.1: CommitteeAlpha::removeMember()

Recommendation Revise the `removeMember()` function to properly update all affected internal states, including `members` and `memberIndexPlusOne`.

Status The issue has been fixed by this commit: `afa3cbf`.

3.2 Suggested Reentrancy Prevention in CommitteeAlpha And Staking

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [6]
- 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` [11] exploit, and the recent `Uniswap/Lendf.Me` hack [10].

We notice there are a number of occasion where the `checks-effects-interactions` principle is violated. Using the `Buyer` as an example, the `deposit()` 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 (lines 128 – 130) starts before effecting the update on internal states (line 131), 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.

```
116     function confirmPayoutStartRequest(uint256 requestIndex_, uint256 payoutId_)
            external {
117         PayoutStartRequest storage request = payoutStartRequests[requestIndex_];
118
119         require(isMember(msg.sender), "Requires member");
120         require(!request.votes[msg.sender], "Already voted");
121         require(!request.executed, "Already executed");
122         require(!isPayoutStartRequestExpired(requestIndex_), "Already expired");
123
124         request.votes[msg.sender] = true;
125         request.voteCount = request.voteCount.add(1);
126
127         if (request.voteCount >= commiteeVoteThreshod) {
```

```
128             ISeller(registry.seller()).startPayout(request.assetIndex, payoutId_);
129             IGuarantor(registry.guarantor()).startPayout(request.assetIndex, payoutId_);
130             IStaking(registry.staking()).startPayout(payoutId_);
131             request.executed = true;
132         }
133     }
```

Listing 3.2: CommitteeAlpha::confirmPayoutStartRequest()

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 `CommitteeAlpha::confirmPayoutAmountRequest()`, `Staking::deposit()`, and `Staking::withdrawReady()`.

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

Status The issue has been fixed by this commit: `318da4d`.

3.3 Potential Bypass Of Staking Withdraw Waiting

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `Staking`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

In order to engage protocol users, the `Tidal` protocol has developed a `Staking` contract to incentivize protocol users. To do that, it naturally supports two staking-related functions `deposit()` and `withdraw()`. In the meantime, the protocol imposes certain lockup time for staking users and the lockup time is managed by the system parameter `withdrawWaitTime`, which has the default 14 days.

While examining the lockup logic, we notice its effectiveness may be significantly reduced if the user makes several consecutive withdraw requests with the total amount larger than the user balance. In other words, the withdraw request may be pre-arranged to avoid the lockup time period. To mitigate, there is a need to ensure the sum of requested amount for withdrawal should not be larger than the user balance.

```
187     function withdraw(uint256 amount_) external {
188         require(!hasPendingPayout(), "Has pending payout");
189
```

```
190         uint256 userAmount = balanceOf(msg.sender);
191         require(userAmount >= amount_, "Not enough amount");
192
193         withdrawRequestMap[msg.sender].push(WithdrawRequest({
194             time: getCurrentWeek(),
195             amount: amount_,
196             executed: false
197         }));
198
199         emit Withdraw(msg.sender, amount_);
200     }
```

Listing 3.3: Staking::withdraw()

Recommendation Enforce the withdraw lockup by ensuring only legitimate request amount will be honored.

Status The issue has been fixed by this commit: `4588aec`.

3.4 Block Time-Related votingPeriod Adjustment

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GovernorAlpha`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

The `Tidal` protocol has an integrated `GovernorAlpha` contract that is forked from `Compound` to support the submission and vote of proposals. For each submitted proposal, there is a voting period that specifies the duration of voting on a proposal. By design, it uses the number of elapsed blocks to measure the voting duration.

We should mention that different blockchains have different block-producing time. For example, the original `Compound` protocol runs on `Ethereum`, which is assumed to have 15 seconds per block. The audited protocol is expected to be deployed on `Polygon`, which has 2 seconds per block. (The `BSC` chain has the 3 seconds per block.)

To elaborate, we show below the `votingPeriod()` routine. Current implementation assumes `15s` per block, which needs to be revised to be 2 seconds per block since the deployment will be on `Polygon`.

```
36     /// @notice The duration of voting on a proposal, in blocks
```

```
37     function votingPeriod() public pure returns (uint) { return 17280; } // ~3 days in
           blocks (assuming 15s blocks)
```

Listing 3.4: GovernorAlpha::votingPeriod()

Recommendation Accommodate different blockchains for different block time in the above `votingPeriod()` function.

Status The issue has been fixed by this commit: `0eaf4f0`.

4 Conclusion

In this audit, we have analyzed the Tidal Governance design and implementation. The system presents a unique, robust offering as a decentralized non-custodial discretionary mutual cover protocol that offers the DeFi community the ability to hedge against the failure of any DeFi protocol or asset. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.