SMART CONTRACT AUDIT REPORT

for

TIDAL

Prepared By: Shuxiao Wang

PeckShield

May 1, 2021

Document Properties

Client	Tidal
Title	Smart Contract Audit Report
Target	Tidal
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 1, 2021	Xuxian Jiang	Final Release
1.0-rc1	April 25, 2021	Xuxian Jiang	Release Candidate #1
0.2	April 23, 2021	Xuxian Jiang	Add More Findings #1
0.1	April 19, 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

Given the opportunity to review the **Tidal** 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.

The basic information of Tidal is as follows:

Table 1.1: Basic Information of Tidal

Item	Description
Issuer	Tidal
Website	https://tidal.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 1, 2021

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

in this audit. Note that Tidal 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 (ae26c14)

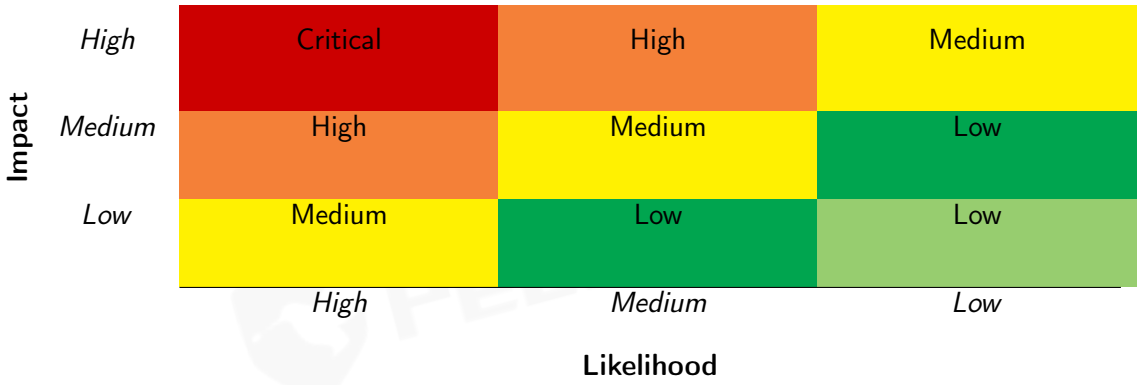
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/TidalFinance/tidal-contracts.git (c2fdb83)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

- 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) [13], 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 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	1	■
Medium	2	■■
Low	3	■■■
Informational	2	■■
Total	8	

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 high-severity vulnerability, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 2 informational recommendation.

Table 2.1: Key Tidal Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Re-Architecture Of Common Parameters And Configurations	Coding Practices	Fixed
PVE-002	High	Permissionless setOffset() And setExtra()	Security Features	Fixed
PVE-003	Informational	Redundant State/Code Removal	Coding Practices	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-005	Medium	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-006	Low	Improved Boundary Case Handling In Buyer::withdraw()	Business Logic	Fixed
PVE-007	Low	Input Validation In Guarantor::finishPayout()	Security Features	Fixed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	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 Re-Architecture Of Common Parameters And Configurations

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [1]

Description

The `Tidal` protocol has a number of essential contracts for different functionalities and duties: `AssetManager`, `Bonus`, `Buyer`, `Guarantor`, and `Seller`. Our analysis with these essential contracts shows that they share a number of configurations, parameters, and functions. These shared states and functions are better relocated to a dedicated registry or address provider.

To elaborate, we use `Bonus` and `Buyer` contracts as an example. Both contracts have defined the states `tidalToken`, `seller`, and `guarantor`, as well as their `setter` routines as shown below.

```
38      function setTidalToken(IERC20 tidalToken_) external onlyOwner {
39          tidalToken = tidalToken_;
40      }
41
42      function setBuyer(IBuyer buyer_) external onlyOwner {
43          buyer = buyer_;
44      }
45
46      function setSeller(ISeller seller_) external onlyOwner {
47          seller = seller_;
48      }
49
50      function setGuarantor(IGuarantor guarantor_) external onlyOwner {
51          guarantor = guarantor_;
```

```
52      }
```

Listing 3.1: A number of setters in Bonus

```
95      function setTidalToken(IERC20 tidalToken_) external onlyOwner {
96          tidalToken = tidalToken_;
97      }
98
99      function setAssetManager(IAssetManager assetManager_) external onlyOwner {
100         assetManager = assetManager_;
101     }
102
103     function setBonus(IBonus bonus_) external onlyOwner {
104         bonus = bonus_;
105     }
106
107     function setSeller(ISeller seller_) external onlyOwner {
108         seller = seller_;
109     }
110
111     function setGuarantor(IGuarantor guarantor_) external onlyOwner {
112         guarantor = guarantor_;
113     }
114
115     function setPlatform(address platform_) external onlyOwner {
116         platform = platform_;
117     }
```

Listing 3.2: A number of setters in Buyer

Apparently, the scattered duplicates of these states and their setters bring additional operation overhead and may cause inconsistency. From the maintenance and protocol consistency perspective, it is strongly suggested to relocate these common states and functions to a dedicated registry-style contract.

Recommendation Re-architect current design to have a registry contract with common states and routines.

Status The issue has been fixed by this commit: `fe1ef6c`.

3.2 Permissionless setOffset() And setExtra()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `WeekManaged`
- Category: Security Features [8]
- CWE subcategory: CWE-282 [2]

Description

In `Tidal`, a number of essential contracts require seamless coordination on the dissemination (and claim) of realized premium and bonus. The coordination is facilitated with the `WeekManaged` contract. In the following, we examine the functionality provided by this `WeekManaged` contract.

To elaborate, we show below the `WeekManaged` contract. It has two core state variables `offset` and `extra`. Both are required to properly calculate `getCurrentWeek()`, which is crucial for the coordination of involved contracts. It comes to our attention both states have their setters and these setters are permissionless, which means they can be modified by anyone!

```
5  contract WeekManaged {

7      uint256 public offset = 4 days;
8      uint256 public extra = 0;

10     function getCurrentWeek() public view returns(uint256) {
11         return (now + offset + extra) / (7 days);
12     }

14     function getNow() public view returns(uint256) {
15         return now + extra;
16     }

18     function getUnlockWeek() public view returns(uint256) {
19         return getCurrentWeek() + 2;
20     }

22     function getUnlockTime(uint256 time_) public view returns(uint256) {
23         require(time_ + offset > (7 days), "Time not large enough");
24         return ((time_ + offset) / (7 days) + 2) * (7 days) - offset;
25     }

27     function setOffset(uint256 offset_) external {
28         offset = offset_;
29     }

31     function setExtra(uint256 extra_) external {
32         extra = extra_;
33     }
```

```
34 }
```

Listing 3.3: The WeekManaged contract

Recommendation Revise these two setters to be permissioned to ensure proper coordination of `Tidal` contracts, especially `Bonus`, `Buyer`, `Guarantor`, and `Seller`.

Status The issue has been fixed by this commit: `186f9c2`.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

Description

Tidal makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Ownable`, to facilitate its code implementation and organization. For example, the `Buyer` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `MAXIMUM_ITERATION` state variable in `Seller`, this variable is not used anywhere. Note that the same state variable is also defined in `Guarantor` and is not used either.

```
10 contract AssetManager is IAssetManager, Ownable {
11
12     struct Asset {
13         address token;
14         uint8 category;  // 0 - low, 1 - medium, 2 - high
15         bool deprecated;
16     }
17     ...
18 }
```

Listing 3.4: The Asset Data Structure in AssetManager

Moreover, there is a data structure `Asset` defined in `AssetManager`, which contains a member field `deprecated`. The field is designed to indicate whether the associated token is deprecated. However, an examination on current logic shows this field is currently not used.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: dee1520.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [9]
- 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 approve() routine and analyze 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. On its entry of approve(), there is a requirement, i.e., require(!((_value != 0) && (allowed[msg.sender][_spender] != 0))). This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling approve(_spender, 0)) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known approve()/transferFrom() race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201         // To change the approve amount you first have to reduce the addresses`
202         //  allowance to zero by calling `approve(_spender, 0)` if it is not
203         //  already 0 to mitigate the race condition described here:
204         //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.5: USDT Token Contract

Because of that, a normal call to approve() with a currently non-zero allowance may fail. In the following, we use the Buyer::beforeUpdate() routine as an example. This routine is designed to approve a specific token for swap contract. To accommodate the specific idiosyncrasy, there is a

need to approve() twice (lines 200 – 201): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
171     // Called every week.
172     function beforeUpdate() public {
173         uint256 currentWeek = getCurrentWeek();
175         require(weekToUpdate < currentWeek, "Already called");
177         if (weekToUpdate > 0) {
178             uint256 totalForGuarantor = 0;
179             uint256 totalForSeller = 0;
180             uint256 feeForPlatform = 0;
182             // To preserve last week's data before update buyers.
183             for (uint16 index = 0; index < assetManager.getAssetLength(); ++index) {
184                 uint256 premiumOfAsset = assetSubscription[index].mul(
185                     getPremiumRate(index)).div(PREMIUM_BASE);
187                 premiumForGuarantor[index] = premiumOfAsset.mul(guarantorPercentage).div
                        (PERCENTAGE_BASE);
188                 totalForGuarantor = totalForGuarantor.add(premiumForGuarantor[index]);
190                 feeForPlatform = feeForPlatform.add(premiumOfAsset.mul(
                        platformPercentage).div(PERCENTAGE_BASE));
192                 premiumForSeller[index] = premiumOfAsset.mul(PERCENTAGE_BASE.sub(
                        guarantorPercentage).sub(platformPercentage)).div(PERCENTAGE_BASE);
193                 totalForSeller = totalForSeller.add(premiumForSeller[index]);
195                 // Calculate assetUtilization from assetSubscription and seller.
                        assetBalance
196                 assetUtilization[index] = getUtilization(index);
197             }
199             IERC20(baseToken).transfer(platform, feeForPlatform);
200             IERC20(baseToken).approve(address(guarantor), totalForGuarantor);
201             IERC20(baseToken).approve(address(seller), totalForSeller);
202         }
204         weekToUpdate = currentWeek;
205     }
```

Listing 3.6: Buyer::beforeUpdate()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the transfer() function does not have a return value. However, the IERC20 interface has defined the transfer() interface with a bool return value. 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

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

Status The issue has been fixed by this commit: `73ee96a`.

3.5 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [5]

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` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

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 (line 276) starts before effecting the update on internal states (line 277), 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.

```
274     // Deposit
275     function deposit(uint256 amount_) external {
276         baseToken.safeTransferFrom(msg.sender, address(this), amount_);
```

```
277         userInfoMap[msg.sender].balance = userInfoMap[msg.sender].balance.add(amount_);
278     }
279
280     // Withdraw
281     function withdraw(uint256 amount_) external {
282         require(userInfoMap[msg.sender].balance > amount_, "not enough balance");
283         baseToken.safeTransfer(msg.sender, amount_);
284         userInfoMap[msg.sender].balance = userInfoMap[msg.sender].balance.sub(amount_);
285     }
```

Listing 3.7: Buyer::deposit()

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 `deposit()`, `withdraw()`, `claimBonus()` in `Buyer`, `deposit()`, `reduceDeposit()`, `withdrawReady()`, `claimPremium()`, `claimBonus()`, and `finishPayout()` in both `Guarantor` and `Seller`. The adherence of `checks-effects-interactions` best practice in these routines is strongly recommended.

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

Status The issue has been fixed by this commit: `b9f57d0`.

3.6 Improved Boundary Case Handling In Buyer::withdraw()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Buyer`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

As mentioned earlier, `Tidal` primarily consists of insurance purchasers and sellers. Since pure peer-to-peer matching platforms on an individual bases have failed to gain traction in areas related to both lending and insurance, `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. In the following, we report an issue on the `withdraw` logic for insurance purchasers.

To elaborate, we show below the responsible `withdraw()` routine. Its logic is rather straightforward in deducting the withdrawn amount from the internal record and transfer the funds back to the user.

However, the imposed requirement prevents the user from withdrawing all entitled funds. Specifically, the requirement `require(userInfoMap[msg.sender].balance > amount_)` should be revised as `require(userInfoMap[msg.sender].balance >= amount_)`, so that a full withdrawal is possible.

```
280     // Withdraw
281     function withdraw(uint256 amount_) external {
282         require(userInfoMap[msg.sender].balance > amount_, "not enough balance");
283         baseToken.safeTransfer(msg.sender, amount_);
284         userInfoMap[msg.sender].balance = userInfoMap[msg.sender].balance.sub(amount_);
285     }
```

Listing 3.8: Buyer::withdraw()

Recommendation Allow for full withdarawl for insurance buyers.

Status The issue has been fixed by this commit: `188b1a1`.

3.7 Input Validation In Guarantor::finishPayout()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Guarantor, Seller`
- Category: Time and State [12]
- CWE subcategory: CWE-682 [6]

Description

`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. For each claim, there is a unique `payoutId` to keep track of the claim and payment status. During the analysis of the lifecycle of a specific `payoutId`, we notice a specific routine `finishPayout()` (in both `Guarantor` and `Seller`). This routine can be improved to validate the given `payoutId` is legitimate.

To elaborate, we show below this routine. Note this payout information of a specific `payoutId` is saved in a mapping structure `payoutInfo`. This routine is permissionless and simply trusts the given `payoutId`. In other words, if a non-present or future `payoutId` is given, the logic may still update the `finished` state to be `true` (line 307). Note that the `Seller` contract shares the same issue.

```
296     function finishPayout(uint256 payoutId_) external {
297         require(!payoutInfo[payoutId_].finished, "already finished");
298
299         if (payoutInfo[payoutId_].paid < payoutInfo[payoutId_].total) {
300             // In case there is still small error.
301             IERC20(baseToken).safeTransferFrom(msg.sender, address(this), payoutInfo[
                    payoutId_].total - payoutInfo[payoutId_].paid);
```

```
302             payoutInfo[payoutId_].paid = payoutInfo[payoutId_].total;
303         }
304
305         IERC20(tidalToken).safeTransfer(payoutInfo[payoutId_].toAddress, payoutInfo[
                payoutId_].total);
306
307         payoutInfo[payoutId_].finished = true;
308     }
```

Listing 3.9: Guarantor::finishPayout()

Recommendation Validate the input argument in `finishPayout()` by filtering our invalid requests.

Status The issue has been fixed by this commit: `0ff13ebc`.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

In the `Tidal` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., factory contract whitelisting, fee adjustment, and parameter setting). It also has the privilege to regulate or govern the flow of assets for borrowing and lending among the involved components, i.e., `AssetManger`, `Bonus`, and `Buyer`.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Tidal` protocol.

```
105     function setBaseToken(IERC20 baseToken_) external onlyOwner {
106         baseToken = baseToken_;
107     }

109     function setTidalToken(IERC20 tidalToken_) external onlyOwner {
110         tidalToken = tidalToken_;
111     }

113     function setAssetManager(IAssetManager assetManager_) external onlyOwner {
114         assetManager = assetManager_;
115     }
```

```
117     function setBuyer(IBuyer buyer_) external onlyOwner {
118         buyer = buyer_;
119     }

121     function setBonus(IBonus bonus_) external onlyOwner {
122         bonus = bonus_;
123     }
```

Listing 3.10: Various Setters in Seller

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance is currently managed by a `timelock` mechanism. Note the `timelock` still has a privileged admin account.

We point out that a compromised `timelock's admin` account would allow the attacker to add a malicious token or change other settings to compromise funds, which directly undermines the assumption of the `Tidal` protocol.

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 fixed by this commit: `440286d`.

4 Conclusion

In this audit, we have analyzed the Tidal 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-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[6] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[7] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[8] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

[12] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[13] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[14] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[15] PeckShield. PeckShield Inc. https://www.peckshield.com.

[16] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[17] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.