SMART CONTRACT AUDIT REPORT

for

HAKKA FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China
September 10, 2020

Document Properties

Client	Hakka Finance
Title	Smart Contract Audit Report
Target	3FMutual
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	September 10, 2020	Xuxian Jiang	Final Release
0.2	September 3, 2020	Xuxian Jiang	Additional Findings
0.1	September 2, 2020	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 **3FMutual** design document and related smart contract source code, we in the report outline 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 3FMutual

Third Floor Mutual (3FMutual) is an insurance agreement that runs on blockchain and aims to provide a hedging mechanism relative to DeFi products, such as `MakerDAO`. In other words, it functions as a rainy day fund like mechanism which helps you to hedge against collapse risk of chosen DeFi protocols. However, it is neither an option nor a short position of `ETH/DAI/MKR`. The rainy day fund like design means it's more like collective insurance. It certainly recognizes the rising value of assets locked in DeFi. In the meantime, it also exposes the fragile side from DeFi-related attacks/risks. In essence, the proposition of 3FMutual allows DeFi users to hedge these risks with a collective insurance.

The basic information of 3FMutual is as follows:

Table 1.1: Basic Information of 3FMutual

Item	Description
Issuer	Hakka Finance
Website	https://hakka.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 10, 2020

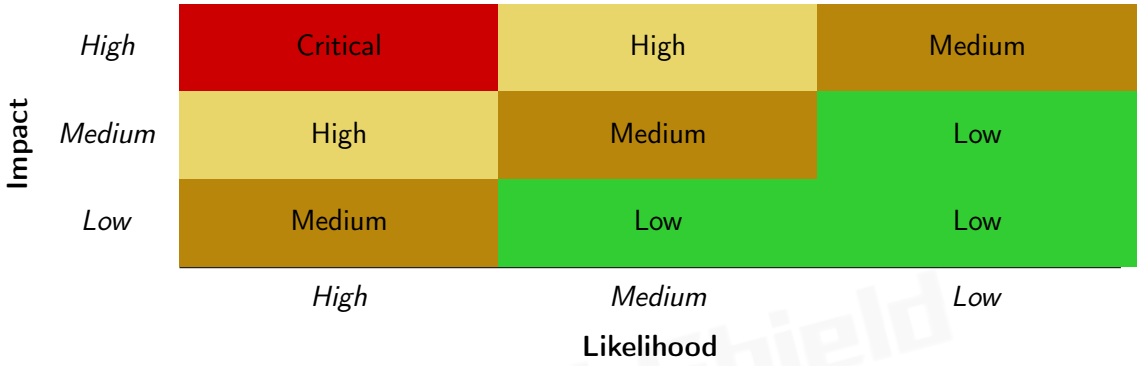
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/artistic709/3FMutual_audit (4fd6c5f)

1.2 About PeckShield

PeckShield Inc. [18] 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 OWASP Risk Rating Methodology [13]:

- 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.

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

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 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) [12], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 an 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 3FMutual implementation. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	1	■
Total	5	

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, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Integer Overflow in *sqrt()*	Numeric Errors	Confirmed
PVE-002	Low	Better Handling of Ownership Transfers	Security Features	Fixed
PVE-003	Medium	Proper Asset Returns After Global Settlement	Business Logics	Fixed
PVE-004	Low	Unhandled Dust in distributeEx()	Coding Practices	Confirmed
PVE-005	Informational	Removal of Expired Insurance Units	Time and State	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Possible Integer Overflow in *sqrt()*

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Underwriter/SafeMath`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [2]

Description

In 3FMutual, depositing `ETH`s to buy the collective insurance requires the calculation of resulting share, which necessitates the familiar `sqrt()` function in order to calculate the integer square root of a given number. The `sqrt()` function, implemented in `SafeMath`, follows the `Babylonian` method for calculating the integer square root. Specifically, for a given x, we need to find out the largest integer z such that $z^2 <= x$.

```
33      function sqrt(uint256 x)
34          internal
35          pure
36          returns (uint256 y)
37      {
38          uint256 z = ((add(x, 1)) / 2);
39          y = x;
40          while (z < y)
41          {
42              y = z;
43              z = ((add((x / z), z)) / 2);
44          }
45      }
```

Listing 3.1: underwriter.sol

The current `sqrt()` implementation is shown above. The initial value of z to the iteration was given as $z = ((add(x, 1))/2)$, which results in an integer overflow when $x = max\_int256 =$

$int256(2 ** 255 - 1)$. In other words, the overflow essentially sets z to zero, leading to a `division by zero` in the calculation of $z = ((add((x/z), z))/2)$ (line 77).

Note that this does not result in an incorrect return value from `sqrt()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a `divide by zero`, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss.

To address this particular corner case, We suggest to change the initial value to $z = add(x/2, 1)$, making `sqrt()` well defined over its all possible inputs.

Recommendation Revise the above calculation to avoid the unnecessary integer overflow.

```
33    function sqrt(uint256 x)
34        internal
35        pure
36        returns (uint256 y)
37    {
38        uint256 z = add(x >> 1, 1);
39        y = x;
40        while (z < y)
41        {
42            y = z;
43            z = ((add((x / z), z)) / 2);
44        }
45    }
```

Listing 3.2: underwriter.sol

Status This issue has been confirmed. Considering the fact this contact has been deployed and this cover case poses no real damage, the team decides to leave it as is for the time being.

3.2 Better Handling of Ownership Transfers

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `InsuranceImprovementProposalProxy/Ownable`
- Category: Security Features [7]
- CWE subcategory: CWE-282 [3]

Description

The `Ownable` smart contract implements a rather basic access control mechanism that allows a privileged account, i.e., `owner`, to be granted exclusive access to typically sensitive functions (e.g., the

setting of certain risk parameters). Because of the owner-level access and the implications of these sensitive functions, the owner account is critical for the 3FMutual security.

Within this contract, a specific function, i.e., transferOwnership(), allows for the ownership update. However, current implementation achieves its goal within a single transaction. This is reasonable under the assumption that the newOwner parameter is always correctly provided. However, in the unlikely situation, when an incorrect newOwner is provided, the contract ownership may be forever lost, which would be devastating for the entire system operation and maintenance.

As a common best practice, instead of achieving the owner update within a single transaction, it is suggested to split the operation into two steps. The first step initiates the owner update intent and the second step accepts and materializes the update. These two steps should be executed in two separate transactions. By doing so, it can greatly alleviate the concern of accidentally transferring the contract ownership to an uncontrolled address. In other words, this two-step procedure ensures that an owner's public key cannot be nominated unless there is an entity that has the corresponding private key. This is explicitly designed to prevent unintentional errors in the owner transfer process.

```
17      function transferOwnership(address newOwner) public onlyOwner {
18          require(newOwner != address(0), "invalid address");
19          emit OwnershipTransferred(owner, newOwner);
20          owner = newOwner;
21      }
```

Listing 3.3: IIP.sol

Recommendation As suggested, the ownership transition can be better managed with a two-step approach, such as, using these two functions: transferOwnership() and acceptOwnership(). Specifically, the changeOwner() function keeps the new address in the storage, _newOwner, instead of modifying the _owner directly. The acceptOwner() function checks whether _newOwner is msg.sender to ensure that _newOwner signs the transaction and verifies herself as the new owner. Only after the successful verification, _newOwner would effectively become the _owner.

```
17      function transferOwnership(address _newOwner) internal {
18          require(_newOwner != address(0), "Owner should not be 0 address");
19          require(_newOwner != owner, "The current and new owner cannot be the same");
20          require(_newOwner != newOwner, "Cannot set the candidate owner to the same
                address");
21          newOwner = _newOwner;
22      }

24      function acceptOwnership() public {
25          require(msg.sender == newOwner, "msg.sender and _newOwner must be the same");
26          emit OwnershipTransferred(owner, newOwner);
27          owner = newOwner;
28          newOwner = address(0);
29      }
```

Listing 3.4: IIP.sol (revised)

Status This issue has been fixed by this commit: 0da1a7f1406e0397617b139c2ba93c14e9c31f7e.

3.3 Proper Asset Returns After Global Settlement

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `3FMutual`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [6]

Description

`ThreeFMutual` is the main contract that provides the collective insurance for users to hedge risks. In particular, users can buy insurance via a number of `buy()` variants. One variant is the use of the contract's fallback function.

For elaboration, we show its code snippet below. We notice that the logic of returning funds after the global settlement of `MakerDAO`, i.e., `tick()==true`, needs to be re-visited. Specifically, if there is a global settlement of `MakerDAO`, any new fund transferred into 3FMutual needs to be properly returned back. The current prototype assumes the sender is a contract wallet. However, the sender might an EOA (Externally Owned Account) as well.

```
343     // contract wallets, sorry insurance only for human
344     function buy()
345         public
346         payable
347     {
348         // ticker
349         if(tick()) {
350             sendContract(msg.sender, msg.value);
351             return;
352         }
353
354         buyCore(msg.sender, msg.value, 0, address(0));
355     }
356
357     // fallback
358     function () external payable {
359         buy();
360     }
```

Listing 3.5: 3FMutual.sol

With that, we need to differentiate the sender type: if it is a contract wallet, we do not need to change with the current `sendContract()` helper; if it is an EOA, we need to invoke another helper,

i.e., sendHuman().

Recommendation Revise the above asset-returning logic as follows.

```
343     // contract wallets, sorry insurance only for human
344     function buy()
345         public
346         payable
347     {
348         // ticker
349         if(tick()) {
350             if(msg.sender == tx.origin)
351                 sendHuman(msg.sender, msg.value);
352             else
353                 sendContract(msg.sender, msg.value);
354             return;
355         }
356
357         buyCore(msg.sender, msg.value, 0, address(0));
358     }
359
360     // fallback
361     function () external payable {
362         buy();
363     }
```

Listing 3.6: 3FMutual.sol

Status This issue has been fixed by this commit: 0da1a7f1406e0397617b139c2ba93c14e9c31f7e.

3.4 Unhandled Dust in distributeEx()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: 3FMutual
- Category: Coding Practices [9]
- CWE subcategory: CWE-627 [5]

Description

In 3FMutual, the insurance-buying payments are distributed into four different categories: dividend , IIP, Hakka, and the insurance pool. The first category dividend shares the payments among all current buyers (and the portion of earlier buyers' insurances are also reduced); the second category IIP chooses to fund insurance improvement proposals (IIPs); the third category contributes back to

the Hakka foundation; and the last category retains the majority of funds in insurance pool to fulfill possible withdraws. Their shares are shown in the following table:

Table 3.1: The Distribution of Insurance-Buying Payments

Destination	Percentage	Note
Dividend	15%	Distributed equally to all share holders
IIP	10%	Distributed to fund Insurance Improvement Proposals (IIPs)
Pool	55%	Distributed to the insurance pool
Hakka	20%	Distributed to the Hakka foundation

The distribution logic is implemented with two helper routines, i.e., distributeEx() and distributeIn(). The former handles the distribution to external stakeholders, including Hakka and IIPs while the latter distributes to current buyers and the insurance pool. Notice that any dust resulted from rounding issues is designed to be collected into the insurance pool.

```
420     /**
421      * @dev pay external stakeholder
422      */
423     function distributeEx(uint256 _eth, address payable _agent) internal {
424         // 20% to external
425         uint256 ex = _eth / 5 ;
426
427         // 10% to IIP
428         uint256 _iip = _eth / 10;
429
430         if(player[_agent].isAgent){
431             uint256 refRate = player[_agent].level.add(6);
432             uint256 _ref = _eth.mul(refRate) / 100;
433             player[_agent].ref = player[_agent].ref.add(_ref);
434             player[_agent].accumulatedRef = player[_agent].accumulatedRef.add(_ref);
435             ex = ex.sub(_ref);
436         }
437
438         sendContract(IIP, _iip);
439         sendContract(hakka, ex);
440     }
441
442     /**
443      * @dev Distribute to internal
444      */
445     function distributeIn(address _buyer, uint256 _eth, uint256 _shares) internal {
446         // 15% to share holder
447         uint256 _div = _eth.mul(3) / 20;
448
449         // 55% to insurance pool
450         uint256 _pool = _eth.mul(55) / 100;
451
452         // distribute dividend share and collect dust
```

```
453         uint256 _dust = updateMasks(_buyer, _div, _shares);

455         // add eth to pool
456         pool = pool.add(_dust).add(_pool);


459     }
```

Listing 3.7: 3FMutual.sol

Our analysis shows that `distributeIn()` has properly collected the dust into the `insurance pool` (line 456), but not `distributeEx()`. Lastly, we notice the above share percentages are not the same as the official Medium article: https://medium.com/hakkafinance/3f-mutual-insurance-480bfb30a30d. It is strongly suggested to keep them consistent.

Recommendation Revise the logic to collect the dust, if any, from `distributeEx()` into the `insurance pool` as well.

Status This issue has been confirmed. Note that the dust might be less than 100 `WEI`. The team considers it likely gas-inefficient to collect the dust and thus chooses to leave it as is.

3.5 Removal of Expired Insurance Units

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `3FMutual`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [4]

Description

3FMutual maintains an internal record `unitToExpire` to keep track of the insurance units due at s particular day. This number is increased (in `buyCore()`) when there is a new insurance payment and it is supposed to be decreased when an earlier insurance is expired. This number is important as it can be used to monitor the insurance dynamics.

In the following, we show the code snippet of `updatePlayerUnit()`. The logic updates the insurance units of a buyer and will be invoked in two occasions: the first one is the new insurance payment and the second one is when a beneficiary claims the owned share of the insurance pool.

Our analysis shows that the update logic can be improved. In particular, note that 3FMutual specifies the maximum insurance period, i.e., `maxInsurePeriod = 100`. As a result, we only need to maintain the insurance units due within this period (as insurance units outside this period are all expired). By doing so, we can delete stale states and free unused storage space.

```
404     /**
405      * @dev Update player's units of insurance
406      */
407     function updatePlayerUnit(address _player) internal {
408         uint256 _today = player[_player].plyrLastSeen;
409         uint256 expiredUnit = 0;
410         if(_today != 0){
411             while(_today < today){
412                 expiredUnit = expiredUnit.add(unitToExpirePlayer[_player][_today]);
413                 _today += 1;
414             }
415             player[_player].units = player[_player].units.sub(expiredUnit);
416         }
417         player[_player].plyrLastSeen = today;
418     }
```

Listing 3.8: 3FMutual.sol

Recommendation Revise the `updatePlayerUnit()` logic to remove the unused `unitToExpire` entries (line 413).

```
404     /**
405      * @dev Update player's units of insurance
406      */
407     function updatePlayerUnit(address _player) internal {
408         uint256 _today = player[_player].plyrLastSeen;
409         uint256 expiredUnit = 0;
410         if(_today != 0){
411             while(_today < today){
412                 expiredUnit = expiredUnit.add(unitToExpirePlayer[_player][_today]);
413                 unitToExpirePlayer[_player][_today] = 0;
414                 _today += 1;
415             }
416             player[_player].units = player[_player].units.sub(expiredUnit);
417         }
418         player[_player].plyrLastSeen = today;
419     }
```

Listing 3.9: 3FMutual.sol

Status This issue has been confirmed. The team takes the approach of directly zeroing out `unitToExpirePlayer[_player][_today]`. Similarly, it is also applicable for `unitToExpire[today]` in `tick()`. The fix can be found in this commit: 0da1a7f1406e0397617b139c2ba93c14e9c31f7e.

3.6 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, we always suggest using fixed compiler version whenever possible. As an example, we highly

encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.10;` instead of `pragma solidity ^0.6.10;`.

Moreover, we strongly suggest not to use experimental Solidity features (e.g., `pragma experimental ABIEncoderV2`) or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries.

Last but not least, it is always important to develop necessary risk control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in the mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the 3FMutual design and implementation. The proposed collective insurance system presents a unique innovation that can be used hedge possible risks against current DeFi protocols. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [14, 15, 16, 17, 19].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [20] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-282: Improper Ownership Management. https://cwe.mitre.org/data/definitions/282.html.

[4] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[5] MITRE. CWE-627: Dynamic Variable Evaluation. https://cwe.mitre.org/data/definitions/627.html.

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

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

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

[14] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[15] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[16] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[17] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[19] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[20] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.