SMART CONTRACT AUDIT REPORT

for

GATHERDAO

Prepared By: Shuxiao Wang

Hangzhou, China

Oct. 15, 2020

Document Properties

Client	GatherDAO
Title	Smart Contract Audit Report
Target	GatherDAO
Version	1.0
Author	Chiachih Wu
Auditors	Chiachih Wu, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Oct. 15, 2020	Chiachih Wu	Final Release
1.0-rc3	Oct. 13, 2020	Jeff Liu	Release Candidate #3
1.0-rc2	Oct. 4, 2020	Jeff Liu	Release Candidate #2
1.0-rc1	Sep. 16, 2020	Jeff Liu	Release Candidate #1
0.3	Sep. 11, 2020	Chiachih Wu	PVE-008 Updated
0.2	Sep. 10, 2020	Chiachih Wu	Additional Findings
0.1	Sep. 9, 2020	Chiachih Wu	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 **GatherDAO** 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. The issues found during the audit have all been promptly addressed by the team, and here in this document we describe in detail our audit results.

1.1 About GatherDAO

GatherDAO attempts to make gathering crypto tokens simple and secure. It uses audited smart contracts for a simple gather, an auction gather, or a bonding price curve gather, etc. The project itself is a DAO which facilitates community-based governance and allows its users to decide the direction of the project.

The basic information of GatherDAO is as follows:

Table 1.1: Basic Information of GatherDAO

Item	Description
Issuer	GatherDAO
Website	https://github.com/GatherMaker
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Oct. 15, 2020

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

- https://github.com/GatherMaker/GatherDAOContract (a757794)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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

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) [6], 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 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 GatherDAO 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. We also measured the gas consumption of key operations with comparison with the popular `UniswapV2`. The purpose here is to not only statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool, but also understand the performance in a realistic setting.

Severity	# of Findings	
Critical	1	■
High	0	
Medium	2	■ ■
Low	3	■ ■ ■
Informational	5	■ ■ ■ ■ ■
Total	11	

Beside the performance measurement, we further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs. So far, we have 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 critical-severity vulnerability, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 5 informational recommendations.

Table 2.1: Key GatherDAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility with Deflationary Tokens	Business Logics	Fixed
PVE-002	Info.	Suggested Adherence of Checks-Effects-Interactions	Business Logics	Fixed
PVE-003	Medium	Possible DoS Attack in Contribute()	Business Logics	Fixed
PVE-004	Critical	Confused Deputy in contributeToPool()	Business Logics	Fixed
PVE-005	Low	Out-of-gas Risks in distributeTokens()	Business Logics	Fixed
PVE-006	Info.	Privileged Interface to Move Out Assets	Business Logics	Fixed
PVE-007	Info.	Unreachable Condition in distributeTokens()	Coding Practices	Fixed
PVE-008	Info.	Redundant Variables and Operations	Coding Practices	Fixed
PVE-009	Low	Duplicate Contributors Due to addAdmin()	Business Logics	Fixed
PVE-010	Info.	Unable to Add/Remove Admins After Pool Initialization	Business Logics	Fixed
PVE-011	Medium	Logic Error in setPoolToRefunded()	Coding Practics	Fixed

Please refer to Section 3 for details.

3 Detailed Results

3.1 Incompatibility with Deflationary Tokens

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

Description

In the `GatherCore` contract, users can use the `contributePoolCurrency()` function to contribute `_amount` of `poolCurrency` into the `GatherCore` contract (line 40). Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of `GatherCore`. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function contributePoolCurrency(uint256 _amount) external {
        require(
            poolCurrency.balanceOf(msg.sender) >= _amount,
            "BALANCE < CONTRIBUTION"
        );
        poolCurrency.safeTransferFrom(msg.sender, address(this), _amount);
        contribute(msg.sender, _amount);
    }
```

Listing 3.1: GatherCore.sol

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `contributePoolCurrency()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of GatherDAO and affects protocol-wide operation and maintenance. Specifically, in line 41, `_amount` is passed into `contribute()` and the book-keeping records associated with `msg.sender` would be updated as follows: `contributorsInfo[msg.sender].balance = contributorsInfo[msg.sender].balance.add(_amount)`. If the `_amount` is not the amount received by `GatherCore` (i.e., part of them are burned), `GatherCore` could have less tokens than the internal asset records.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into GatherDAO for indexing. However, as a trustless intermediary, GatherDAO may not be in the position to effectively regulate the entire process. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation To accommodate the support of possible deflationary tokens, it is better to check the balance before and after the `safeTransferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost.

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.2 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GatherCore`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice there is an occasion the `checks-effects-interactions` principle is violated. The `contributePoolCurrency()` 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 40) starts before effecting the update on internal states (lines 41), hence violating the principle. In this particular case, if the `contribute()` update the state with the total amount (e.g., `contributorsInfo[msg.sender].balance.add(_amount)`) and the external contract has some hidden logic that may be capable of launching `re-entrancy`, a bad actor could drain most of the `poolCurrency` withheld by `GatherCore`.

```
35    function contributePoolCurrency(uint256 _amount) external {
36        require(
37            poolCurrency.balanceOf(msg.sender) >= _amount,
38            "BALANCE < CONTRIBUTION"
39        );
40        poolCurrency.safeTransferFrom(msg.sender, address(this), _amount);
41        contribute(msg.sender, _amount);
42    }
```

Listing 3.2: GatherCore.sol

Specifically, in the case that `poolCurrency` is an ERC777 token, they could hijack a `contributePoolCurrency()` call after `poolCurrency.safeTransferFrom()` in line 40 with a callback function. Within the callback function, they could call the `withdrawContribution()` function to withdraw some `poolCurrency` back with `contributorsInfo[msg.sender]` updated. Now, the control flow goes back to `contributePoolCurrency()` line 41. If the `contribute()` function happens to set `contributorsInfo[msg.sender].balance` with the pre-calculated `contributorsInfo[msg.sender].balance.add(_amount)`, the previous update by `withdrawContribution()` would be flushed, leading to withdrawing tokens out without updating the state. The bad actor could do it again and again until all `poolCurrency` in `GatherCore` are gone because of the unlimited balance. Fortunately, this is not the case.

Recommendation Apply the `checks-effects-interactions` design pattern.

```
35    function contributePoolCurrency(uint256 _amount) external {
36        require(
37            poolCurrency.balanceOf(msg.sender) >= _amount,
38            "BALANCE < CONTRIBUTION"
39        );
40        contribute(msg.sender, _amount);
41        poolCurrency.safeTransferFrom(msg.sender, address(this), _amount);
42    }
```

Listing 3.3: GatherCore.sol

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.3 Possible DoS Attack in Contribute()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GatherCore`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

In the `GatherCore` contract, the internal function `contribute()` updates the states regarding each user's contribution as well as the global states (e.g., `totalPoolContributio`. During each `contribute()` call, the `_value` is checked against `minContribution` (line 60) and `maxContribution` (line 61). Also, the `totalPoolContribution + _value` is not allowed to be greater than `maxPoolAllocation` (line 62).

However, those constraints create an attack surface for denial-of-services. For example, let's say `maxPoolAllocation=100` and `minContribution=10`. When `totalPoolContribution` reaches 80, a bad actor could intentionally `contribute()` 11, which passes all the sanity checks but denies all the following contributions. Specifically, the `totalPoolContribution` after the bad actor's contribution would be $80 + 11 = 91$. No contribution which passes both checks in line 60 and line 62 could be made since $91 + 10 = 101 > 100$.

```
56     function contribute(address _sender, uint256 _value) internal isPoolOpen() {
57         Contributor storage contributor = contributorsInfo[_sender];
58         require(_value > 0, "ZERO_AMOUNT_SENT");
59         uint256 poolContributionSum = totalPoolContribution.add(_value);
60         require(_value >= minContribution, "AMNT < MIN_CNTRBUTN_ALWD");
61         require(_value <= maxContribution, "AMNT > MAX_CNTRBUTN_ALWD");
62         require(
63             poolContributionSum <= maxPoolAllocation,
64             "TOTAL_POOL_CONTRIBUTION_EXCEED"
65         );

67         if (contributor.exists) {
68             uint256 balanceSum = contributor.balance.add(_value);
69             require(
70                 balanceSum <= maxContribution,
71                 "TTL_CNTRBUTN > MX_CNTRBUTN_ALWD"
72             );
73         } else {
74             contributor.exists = true;
75             contributors.push(_sender);
76         }

78         contributor.balance = contributor.balance.add(_value);

80         // Update the total pool value
```

```
81         totalPoolContribution = totalPoolContribution.add(_value);

83         // Update fund raised status
84         updateFundsRaised();

86         emit ContributorContributed(_sender, _value);
87     }
```

Listing 3.4: GatherCore.sol

Even worse, in line 84, `updateFundsRaised()` is invoked to perform the pool state transition. As shown in the code snippet below, the `poolState` is set as `PoolState.CLOSED` if and only if `totalPoolContribution == maxPoolAllocation`.

```
    function updateFundsRaised() internal isPoolOpen() {
        if (totalPoolContribution == maxPoolAllocation) {
            fundsRaised = true;

            poolState = PoolState.CLOSED;

            emit PoolClosed();
        }
    }
```

Listing 3.5: GatherCore.sol

A bad actor could DoS any fund raising activity using GatherDAO by `contribute()` some assets when `totalPoolContribution` is close to `maxPoolAllocation`.

Recommendation Update `minContribution` for the last contribution if necessary.

```
    function contribute(address _sender, uint256 _value) internal isPoolOpen() {
        Contributor storage contributor = contributorsInfo[_sender];
        require(_value > 0, "ZERO_AMOUNT_SENT");
        uint256 poolContributionSum = totalPoolContribution.add(_value);
        require(_value >= minContribution, "AMNT < MIN_CNTRBUTN_ALWD");
        require(_value <= maxContribution, "AMNT > MAX_CNTRBUTN_ALWD");
        require(
            poolContributionSum <= maxPoolAllocation,
            "TOTAL_POOL_CONTRIBUTION_EXCEED"
        );

        if (contributor.exists) {
            uint256 balanceSum = contributor.balance.add(_value);
            require(
                balanceSum <= maxContribution,
                "TTL_CNTRBUTN > MX_CNTRBUTN_ALWD"
            );
        } else {
            contributor.exists = true;
            contributors.push(_sender);
        }
```

```
        contributor.balance = contributor.balance.add(_value);

        // Update the total pool value
        totalPoolContribution = poolContributionSum;

        // Update minContribution for the last contributor
        minContribution = minContribution <= maxPoolAllocation.sub(totalPoolContribution
            ) ? minContribution : maxPoolAllocation.sub(totalPoolContribution);

        // Update fund raised status
        updateFundsRaised();

        emit ContributorContributed(_sender, _value);
    }
```

Listing 3.6: GatherCore.sol

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.4 Confused Deputy in contributeToPool()

- ID: PVE-004
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `GatherCore`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

In the `GatherCore` contract, the `isPoolCurrency` global variable indicates if the current pool uses an ERC20 or ETH as the currency. When `isPoolCurrency`, the corresponding ERC20 address is set at `poolCurrency`. All the following asset movements such as `contributePoolCurrency()` and `sendPoolCurrencyForRefund()` rely on the ERC20 contract deployed at `poolCurrency`.

Based on that, there're two ways for contributors to transfer assets into the pool. If the pool uses ERC20, a contributor should invoke `contributePoolCurrency()`. If the pool uses ETH, `contributeToPool()` should be called. Here, we identified a confused deputy issue in both of them. As shown in the code snippet below, both functions collect the assets, implicitly (ETH) or explicitly (ERC20), before calling `contribute()` to update the internal asset records. However, there's no sanity check to ensure that the current pool is configured with the asset that the `msg.sender` is sending in.

```
    function contributePoolCurrency(uint256 _amount) external {
        require(
            poolCurrency.balanceOf(msg.sender) >= _amount,
            "BALANCE < CONTRIBUTION"
```

```
        );
        poolCurrency.safeTransferFrom(msg.sender, address(this), _amount);
        contribute(msg.sender, _amount);
    }
```

Listing 3.7: GatherCore.sol

```
    function contributeToPool() external payable {
        contribute(msg.sender, msg.value);
    }
```

Listing 3.8: GatherCore.sol

For example, a bad actor could send in 0.001 ETH with `contributeToPool()` to top-up 10^{15} `poolCurrency`. If `poolCurrency` is USDT which has 6 decimals, the bad actor could invoke `withdrawContribution()` right after the `contributeToPool()` call and get 1 millions USDT back. On the other hand, if a bad actor is about to exploit this vulnerability through `contributePoolCurrency()` with a similar trick mentioned above, the `poolCurrency.balanceOf()` and `poolCurrency.safeTransferFrom()` calls fail as the compiler is likely to check the code size of the `poolCurrency` address. Since a pool configured with ETH as the currency has `poolCurrency == address(0)`, the attack through `contributePoolCurrency()` could not happen.

Recommendation Check `isPoolCurrency` in `contributePoolCurrency()` and `contributeToPool()`.

```
    function contributeToPool() external payable {
        require (!isPoolCurrency, "POOL_CONTRIBUTION_MISMATCH");
        contribute(msg.sender, msg.value);
    }
```

Listing 3.9: GatherCore.sol

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.5 Out-of-gas Risks in distributeTokens()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium

- Target: `GatherCore`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

While reviewing the `GatherCore` contract, we noticed that there're multiple functions which walk through the `contributorsInfo[]` array for updating states of each contributor or sending assets to each of them. For example, the `distributeTokens()` function allows the privileged user to send out `erc20Token` to each contributor. However, the transaction to invoke this function is likely out-of-gas when `contributors.length` reaches 400.

```
    function distributeTokens() public onlyPoolOwnerOrAdmin() {
        require(address(erc20Token) != address(0), "TOKEN_ADDRESS_NOT_DEFINED");
        for (uint256 i = 0; i < contributors.length; i++) {
            Contributor storage contributor = contributorsInfo[contributors[i]];
            uint256 allocatedTokens = contributor.tokensAllotted;
            if (allocatedTokens > 0) {
                contributor.tokensAllotted = 0;
                contributor.withdrawnTokens = true;
                if (allocatedTokens != 0) {
                    erc20Token.safeTransfer(contributors[i], allocatedTokens);
                    emit TokensWithdrawn(contributors[i], allocatedTokens);
                }
            }
        }
    }
```

Listing 3.10: GatherCore.sol

Specifically, we know that sending ether from one address to another consumes 21k gas and the total gas limit in one block is around 12m. Based on that, we could perform around 600 ether transfers in one transaction. Even worse, a typical ERC20 transfer costs 36k gas such that `distributeTokens()` is likely to out-of-gas when `contributors.length` is close to 400. Similar issues are identified in `setContributorsTokenAllocation()` and `refundContributions()`.

Recommendation Implement batch processing mechanism to deal with a limited number of contributors in one transaction.

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.6 Privileged Interfaces to Move Out Assets

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GatherCore`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

While reviewing the `GatherCore` contract, we came across a function, `withdrawRandomERC20Token()`, which allows privileged users to tranfser an arbitrary asset owned by `GatherCore` to the `poolOwner` in any `PoolState`. We believe the existence of this function is to recover the ERC20 tokens sent to

`GatherCore` by accident. However, we should not recover the whole balance of `poolCurrency` to prevent internal asset records from being messed up.

```
    function withdrawRandomERC20Token(address _erc20ContractAddress)
        external
        onlyPoolOwnerOrAdmin()
        nonReentrant()
    {
        IERC20 randomToken = IERC20(_erc20ContractAddress);
        uint256 randomTokenBalance = randomToken.balanceOf(address(this));
        randomToken.safeTransfer(poolOwner, randomTokenBalance);
        emit RandomTokenWithdrawlSuccess(poolOwner, randomTokenBalance);
    }
```

Listing 3.11: GatherCore.sol

Recommendation Ensure `_erc20ContractAddress` is not `poolCurrency` in `withdrawRandomERC20Token()`.

```
    function withdrawRandomERC20Token(address _erc20ContractAddress)
        external
        onlyPoolOwnerOrAdmin()
        nonReentrant()
    {
        IERC20 randomToken = IERC20(_erc20ContractAddress);
        require(randomToken != poolCurrency);
        uint256 randomTokenBalance = randomToken.balanceOf(address(this));
        randomToken.safeTransfer(poolOwner, randomTokenBalance);
        emit RandomTokenWithdrawlSuccess(poolOwner, randomTokenBalance);
    }
```

Listing 3.12: GatherCore.sol

Status The issue is addressed by adding the `isPoolRefundedOrCompleted` modifier to `withdrawRandomERC20Token()` function in commit bd47085. To deal with some certain uncovered cases, the `withdrawRandomERC20Token()` is modified to `withdrawRandomAssets()` in commit 56c5dfb.

```
    function withdrawRandomAssets(address _contractAddress, bool withFees)
        external
        isPoolPaidOrRefundedOrCompleted()
        onlyPoolAdmin()
    {
        setPoolToCompleted();
        if (withFees) {
            payOutAdminAndGatherFee();
        }
        uint256 randomBalance;
        if (_contractAddress == address(0)) {
            randomBalance = address(this).balance;
            poolOwner.transfer(randomBalance);
        } else {
            IERC20 randomToken = IERC20(_contractAddress);
```

```
            randomBalance = randomToken.balanceOf(address(this));
            randomToken.safeTransfer(poolOwner, randomBalance);
        }

        emit RandomAssetsWithdrawn(poolOwner, _contractAddress, randomBalance);
    }
```

Listing 3.13: GatherCore.sol

In `withdrawRandomAssets()`, arbitrary erc20s and ether could be withdrew by the `poolOwner` in PAID, REFUNDED, or COMPLETED states. In addition, when `withFees` is set, the fee would be pay to `gatherOwner` and `poolOwner`. Here, we'd like to state that the existence of these functions could still be a concern to contract users, and one way to mitigate is to use multi-sig or timelock mechanisms to execute the privileged functions.

3.7 Unreachable Condition in distributeTokens()

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GatherCore`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

Description

While reviewing the `GatherCore` contract, we identified a unreachable condition in `distributeTokens()`, which could be optimized to reduce gas consumption. Specifically, the sanity check in line 290 is not necessary since `contributor.tokensAllotted` is a `uint256` variable such that the `allocatedTokens != 0` always holds when `allocatedTokens > 0`.

```
282     function distributeTokens() public onlyPoolOwnerOrAdmin() {
283         require(address(erc20Token) != address(0), "TOKEN_ADDRESS_NOT_DEFINED");
284         for (uint256 i = 0; i < contributors.length; i++) {
285             Contributor storage contributor = contributorsInfo[contributors[i]];
286             uint256 allocatedTokens = contributor.tokensAllotted;
287             if (allocatedTokens > 0) {
288                 contributor.tokensAllotted = 0;
289                 contributor.withdrawnTokens = true;
290                 if (allocatedTokens != 0) {
291                     erc20Token.safeTransfer(contributors[i], allocatedTokens);
292                     emit TokensWithdrawn(contributors[i], allocatedTokens);
293                 }
294             }
295         }
296     }
```

Listing 3.14: GatherCore.sol

Recommendation Remove the `if (allocatedTokens != 0)` check as it's not necessary.

```
function distributeTokens() public onlyPoolOwnerOrAdmin() {
    require(address(erc20Token) != address(0), "TOKEN_ADDRESS_NOT_DEFINED");
    for (uint256 i = 0; i < contributors.length; i++) {
        Contributor storage contributor = contributorsInfo[contributors[i]];
        uint256 allocatedTokens = contributor.tokensAllotted;
        if (allocatedTokens > 0) {
            contributor.tokensAllotted = 0;
            contributor.withdrawnTokens = true;
            erc20Token.safeTransfer(contributors[i], allocatedTokens);
            emit TokensWithdrawn(contributors[i], allocatedTokens);
        }
    }
}
```

Listing 3.15: GatherCore.sol

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.8 Redundant Variables and Operations

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GatherDAO`, `GatherCore`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

Description

While reviewing the GatherDAO smart contract, we identified that some variables and opearations are redundant.

Case I The `_poolOwner` variable in `GatherDAO::createPool()` is redundant as it always equals to `msg.sender`.

```
function createPool(
    address payable _poolOwner,
    address _poolCurrencyContractAddress,
    address[] memory _poolAdmins,
    uint256 _maxPoolAllocation,
    uint256 _maxContribution,
    uint256 _minContribution,
    uint256 _gatherFee,
    uint256 _poolFee
) external onlyWhileOpen() {
    require(_poolOwner == msg.sender, "ONLY_POOL_OWNER");

    Pool pool = new Pool(
```

```
            gatherDAOOwner,
            _poolOwner,
            _poolCurrencyContractAddress,
            _poolAdmins,
            _maxPoolAllocation,
            _maxContribution,
            _minContribution,
            _gatherFee,
            _poolFee
        );
        address newPoolAddress = address(pool);
        deployedPools[_poolOwner].push(newPoolAddress);

        emit NewPoolDeployed(newPoolAddress);
    }
```

Listing 3.16: GatherDAO.sol

Recommendation Remove `_poolOwner`, use `msg.sender` instead.

```
    function createPool(
        address _poolCurrencyContractAddress,
        address[] memory _poolAdmins,
        uint256 _maxPoolAllocation,
        uint256 _maxContribution,
        uint256 _minContribution,
        uint256 _gatherFee,
        uint256 _poolFee
    ) external onlyWhileOpen() {
        Pool pool = new Pool(
            gatherDAOOwner,
            msg.sender,
            _poolCurrencyContractAddress,
            _poolAdmins,
            _maxPoolAllocation,
            _maxContribution,
            _minContribution,
            _gatherFee,
            _poolFee
        );
        address newPoolAddress = address(pool);
        deployedPools[msg.sender].push(newPoolAddress);

        emit NewPoolDeployed(newPoolAddress);
    }
```

Listing 3.17: GatherDAO.sol

Case II The `setPoolToOpne()` call (line 136, 150) is redundant since the `poolState` would be set to `REFUNDED` by the first line of code in `refundContributions()` (line 159).

```
130     function sendBalanceForRefund() external payable {
131         require(
```

```
            msg.value >=
                totalPoolContribution.sub(gatherFeeAmount).sub(poolFeeAmount),
            "LESS_ETH_THAN_REQUIRED"
        );
        setPoolToOpen(); // this has owner admin modifier
        refundContributions();
    }
```

Listing 3.18: GatherCore.sol

```
    function sendPoolCurrencyForRefund(uint256 _refundAmount) external {
        require(
            _refundAmount >=
                totalPoolContribution.sub(gatherFeeAmount).sub(poolFeeAmount),
            "LESS_AMOUNT_THAN_REQUIRED"
        );
        setPoolToOpen(); // this has owner admin modifier
        poolCurrency.safeTransferFrom(msg.sender, address(this), _refundAmount);
        refundContributions();
    }
```

Listing 3.19: GatherCore.sol

```
    function refundContributions() public {
        setPoolToRefunded(); // This has admin and open close modifier
        for (uint256 i = 0; i < contributors.length; i++) {
            Contributor storage contributor = contributorsInfo[contributors[i]];
            uint256 contributionBalance = contributor.balance;
            if (contributionBalance > 0) {
                address payable contributorAddress = payable(contributors[i]);
                totalPoolContribution = totalPoolContribution.sub(
                    contributionBalance
                );
                contributor.balance = 0;
                transferFundsGeneric(contributorAddress, contributionBalance);
            }
        }
    }
```

Listing 3.20: GatherCore.sol

Recommendation Remove redundant `setPoolToOpen()` calls. In addition, we suggest to validate the caller to `sendBalanceForRefund()` and `sendPoolCurrencyForRefund()` earlier by `onlyPoolOwnerOrAdmin()`, which makes the code more readable.

```
    function sendBalanceForRefund() external payable onlyPoolOwnerOrAdmin {
        require(
            msg.value >=
                totalPoolContribution.sub(gatherFeeAmount).sub(poolFeeAmount),
            "LESS_ETH_THAN_REQUIRED"
        );
```

```
        refundContributions();
    }
```

Listing 3.21: GatherCore.sol

```
    function sendPoolCurrencyForRefund(uint256 _refundAmount) external
        onlyPoolOwnerOrAdmin {
        require(
            _refundAmount >=
                totalPoolContribution.sub(gatherFeeAmount).sub(poolFeeAmount),
            "LESS_AMOUNT_THAN_REQUIRED"
        );
        poolCurrency.safeTransferFrom(msg.sender, address(this), _refundAmount);
        refundContributions();
    }
```

Listing 3.22: GatherCore.sol

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.9 Duplicate Contributors Due to addAdmin()

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Admin`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

In the `Admin` contract, the internal function `addAdmin()` allows the caller to add `_newAdmin` as one of the privileged users. Whenever a `_newAdmin` is added, three arrays would be updated: `contributorsInfo[]`, `contributors[]`, and `poolAdmins[]`. According to the current implementation, the `_newAdmin` would be added into the `contributors[]` array automatically (line 40) with her related `contributorsInfo` updated (line 38-39). However, there's no sanity check to avoid an existing `_newAdmin` from being added into the `contributors[]` array. Since there're many functions in `GaterCore` traversing the `contributors[]` array, a duplicate contributor is a waste of gas. We suggest to use `contributorsInfo[_newAdmin].exists` to filter out duplicate contributors. This also helps to avoid a duplicate `admin` to be added into `poolAdmins[]`.

```
37    function addAdmin(address _newAdmin) internal {
38        contributorsInfo[_newAdmin].admin = true;
39        contributorsInfo[_newAdmin].exists = true;
40        contributors.push(_newAdmin);
41        poolAdmins.push(_newAdmin);
42    }
```

Listing 3.23: Admin.sol

Recommendation Ensure an existing admin is not added into `contributors[]` and `poolAdmins[]` in `addAdmin()`.

```
function addAdmin(address _newAdmin) internal {
    contributorsInfo[_newAdmin].admin = true;
    if ( !contributorsInfo[_newAdmin].exists ) {
        contributorsInfo[_newAdmin].exists = true;
        contributors.push(_newAdmin);
        poolAdmins.push(_newAdmin);
    }
}
```

Listing 3.24: Admin.sol

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.10 Unable to Add/Remove Admins After Pool Initialization

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Admin`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

Description

While reviewing the `Admin` contract, we noticed that there's no interface to add or remove an admin from a `Pool`. However, there're many cases that we might need to add or remove a privileged user after the initialization process. For example, if one of the admins is compromised or her private key is stolen, there should be a way to revoke her privileges. Note that it's not a good idea to allow each admin to remove another. Otherwise, a malicious one could subvert the whole system. We should have a voting mechanism to allow the majority of the current admins to promote a new candidate or revoke an existing one.

Recommendation Add privileged interfaces for adding/removing an admin with a DAO-like mechanism.

Status The issue is fixed in commit bd470859826f39bfca43e965c9ef93e43676551d

3.11 Logic Error in setPoolToRefunded()

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `State`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1041 [2]

Description

While reviewing the `State` contract, we found a typo/logic error in setPoolToRefunded() when checking the pool state. Line 127 should check if the poolState is either OPEN, CLOSED, or PAID, but in the code it checked if it's not REFUNDED, twice.

```
123     function setPoolToRefunded()
124         public
125         onlyPoolAdmin()
126     {
127         require(poolState != PoolState.REFUNDED || poolState != PoolState.REFUNDED);
```

```
        poolState = PoolState.REFUNDED;
        emit PoolRefunded();
    }
```

Listing 3.25: State.sol

Recommendation Fix the poolState check as following

```
    function setPoolToRefunded()
        public
        onlyPoolAdmin()
    {
        require(poolState == PoolState.OPEN || poolState == PoolState.CLOSED || poolState
            == PoolState.PAID);
        poolState = PoolState.REFUNDED;
        emit PoolRefunded();
    }
```

Listing 3.26: State.sol

Status The issue is fixed in commit a9c5ca039cd7f408e3962e6a83c9be26e255fa95

3.12 Other Suggestions

Since the Solidity language is still maturing and it is common for new compiler versions to include changes that might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.7.0;` instead of `pragma ^0.7.0;`.

Moreover, we strongly suggest not to use experimental Solidity features 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 mainnet.

4 Conclusion

In this audit, we thoroughly analyzed the GatherDAO design and implementation. The system presents a unique design that makes gathering crypto tokens simple and secure. 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 [8, 9, 10, 11, 14].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [16] 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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

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

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

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

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

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

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

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

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