SMART CONTRACT AUDIT REPORT

for

GAINS Vault

Prepared By: Patrick Lou

PeckShield

March 14, 2022

Document Properties

Client	GAINS Associates
Title	Smart Contract Audit Report
Target	GAINS Vault
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Liu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 14, 2022	Xuxian Jiang	Final Release
1.0-rc	March 2, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Patrick Lou
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `GAINS vault` smart contract, 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 GAINS Vault

`GAINS` is one of the biggest and most respected crypto communities, created at the beginning of 2018. The community engages the organization of a number of crypto-related events. During these events, participants can compete for prizes by asking questions or showcasing their knowledge! And community members get the chance to invest in the latest and best crypto projects with terms they would not be able to get as individuals. And the audited contracts implement essential vaults that allow users to deposit (with the protocol-specified lockup period) and withdraw (after the lockup expires). The basic information of audited contracts is as follows:

Table 1.1: Basic Information of GAINS Vault

Item	Description
Name	GAINS Associates
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	March 14, 2022

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

- https://gitlab.com/gains-associates/gains-site/sc-contribution.git (eaa2886)

1.2 About PeckShield

PeckShield Inc. [8] 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 accordingly classified into four categories, 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

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

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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business 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 design and implementation of the `GAINS vault` smart contract, 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 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	1	■
Informational	1	■
Total	3	

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, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation Of Function Arguments	Coding Practices	Confirmed
PVE-002	Informational	Suggested Constant/Immutable Usages For Gas Efficiency	Coding Practices	Confirmed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved Validation Of Function Arguments

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: FlashContributionCollector
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The audited vault contracts are no exception. Specifically, if we examine the FlashContributionCollector contract, it defines a number of public functions, e.g., contribute(), updateTierCaps(), and finalize(), to adjust related configurations. In the following, we show that the related updateTierCaps() function can be benefited from improved validation on the given input arguments.

```
    function updateTierCaps(uint256[] calldata _newTierCaps) external override {
        if (msg.sender != owner) revert Forbidden();
        if (state != State.ACTIVE) revert InactiveContribution();
        for (uint256 _i = 0; _i < _newTierCaps.length; _i++)
            if (contribution.amounts[_i] > _newTierCaps[_i])
                revert InvalidNewTierCaps();
        contribution.tierCaps = _newTierCaps;
        emit TierCapsUpdated(_newTierCaps);
    }
```

Listing 3.1: FlashContributionCollector::updateTierCaps()

Specifically, the function updateTierCaps() is designed to customize the upper bounds of the specified contribution tiers. However, it does not enforce the given _newTierCaps should have the same array length with the tiers, i.e., require(contribution.amounts.length == _newTierCaps.length).

Recommendation Properly revise the above `updateTierCaps()` routine to ensure the given argument has the same array length with `contribution.amounts`.

Status The issue has been confirmed.

3.2 Suggested Constant/Immutable Usages For Gas Efficiency

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Vault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [2]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show a number of key state variables defined in `Vault`, including `token` and `lockingDuration`. If there is no need to dynamically update these key state variables, they can be declared as either constants or `immutable` for gas efficiency. In particular, the state variable `token` can be defined as `immutable` for improved gas efficiency.

```
contract Vault is Ownable {
    using SafeERC20 for IERC20;

    struct Deposit {
        uint256 amount;
        uint256 unlockingTimestamp;
    }

    uint256 public lockingDuration;
    mapping(address => Deposit) public userDeposit;
    address public token;
    ...
```

```
}
```

Listing 3.2: Vault.sol

Recommendation Revisit the state variable definition and make extensive use of `constant`/ `immutable` states.

Status The issue has been confirmed.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Vault`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

Description

In the audited `GAINS vault` smart contract, there exist a certain privileged account `owner` that plays critical roles in governing and regulating the system-wide operations. It also has the privilege to regulate or govern the flow of assets within the protocol. In the following, we show representative privileged operations in the protocol.

```
    function updateLockingDuration(uint256 _duration) external onlyOwner {
        if (_duration == 0) revert InvalidLockingDuration();
        lockingDuration = _duration;
        emit LockingDurationUpdated(_duration);
    }

    function updateUnlockingTimestamp(
        address _account,
        uint256 _unlockingTimestamp
    ) external onlyOwner {
        if (_unlockingTimestamp < block.timestamp)
            revert InvalidUnlockingTimestamp();
        Deposit storage _deposit = userDeposit[_account];
        if (_deposit.amount == 0) revert InvalidAccount();
        if (_unlockingTimestamp > _deposit.unlockingTimestamp)
            revert CannotExtendLocking();
        _deposit.unlockingTimestamp = _unlockingTimestamp;
        emit UnlockingTimestampUpdated(_account, _unlockingTimestamp);
    }
```

Listing 3.3: `Vault::updateLockingDuration()/updateUnlockingTimestamp()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `owner` explicit to `GAINS` users.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `GAINS vault` smart contract, which allows users to deposit (with the protocol-specified lockup period) and withdraw (after the lockup expires). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

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

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.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. PeckShield Inc. https://www.peckshield.com.