SMART CONTRACT AUDIT REPORT

for

Modulus Protocol

Prepared By: Xiaomi Huang

PeckShield

October 6, 2023

Document Properties

Client	Modulus
Title	Smart Contract Audit Report
Target	Modulus
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 6, 2023	Xuxian Jiang	Final Release
1.0-rc	October 1, 2023	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and source code of the `Modulus` protocol, we outline in this 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 contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About Modulus

`Modulus` is a decentralized platform for asymmetric yields distribution generated from staking. It offers the opportunity to earn significantly higher returns while keeping risk exposure controlled. This shift empowers participants to actively engage in a rewarding venture, transforming staking into a dynamic endeavor. Through innovative use of the `Chainlink Verified Random Function (VRF)`, the protocol selects 3 winners weekly based on deposit size and duration, ensuring an equitable opportunity for every participant to secure substantial rewards. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Modulus

Item	Description
Name	Modulus
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 6, 2023

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

- https://github.com/modulusprotocol/Modulus-Protocol-Smart-Contracts.git (5d98a37)

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

- https://github.com/modulusprotocol/Modulus-Protocol-Smart-Contracts.git (850421a)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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 [8]:

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

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

To evaluate the risk, we go through a 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

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

is considered safe regarding the check item. For any discovered issue, we might further deploy contract 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 contract 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 contract 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 contract from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contract, we use the CWE categories in Table 1.4 to classify our findings.

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 Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Modulus` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business 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	2	■ ■
Informational	0	
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. 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, this smart contract is well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Modulus Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Randomness Request in RewardDistributor	Business Logic	Resolved
PVE-002	Low	Possibly OOG For Total Participant Calculation	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 contract is being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Randomness Request in RewardDistributor

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RewardDistributor
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Modulus protocol has a core RewardDistributor contract that is designed to distribute rewards. In the process of reviewing the current logic to make use of Chainlink Verified Random Function (VRF), we notice its implementation may be improved.

To elaborate, we show below the code snippet of the requestRandomness() routine. As the name indicates, this routine is used to request randomness via the use of Chainlink Verified Random Function (VRF). This routine has an argument of randomWordsAmount, which is redundant as the requested number of random words is always 3.


```
    function requestRandomness(address poolAddress,uint256 epochNumber, uint256
        TotalTicketAmount,uint32 randomWordsAmount) external onlyOperator returns(
        uint256){

        address VRFAddress = IRoleRegistry(registryContract).getVRF();
        uint256 requestID = IVRFv2DirectFundingConsumer(VRFAddress).requestRandomWords(
            randomWordsAmount);

        randomnessHistory.push(RandomnessRequest({
            poolAddress: poolAddress,
            epochNumber: epochNumber,
            DrawRequestID: requestID,
            thisDrawTicketAmount: TotalTicketAmount,
            randomWordsAmount: randomWordsAmount,
            fulfilled: false,
            results: new uint256[](3)
```

```
        }));


        uint256 result = randomnessHistory.length - 1;
        return result;
    }
```

Listing 3.1: `RewardDistributor::requestRandomness()`

Recommendation Revise the above `requestRandomness()` logic to ensure only 3 random words are requested.

Status The issue has been fixed by the following commit: `afb8247`.

3.2 Possibly OOG For Total Participants Calculation

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `stETHPool`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Modulus` protocol is designed to support a variety of pools. While examining a specific pool, i.e., `stETHPool`, we notice the calculation of total number of participants may consume too much gas, leading to possible `OOG` issue.

To elaborate, we show below the code snippet of the `sumUptotalParticipant()` routine. As the name indicates, this routine is used to count total participants. It basically iterates the whole set of users (`amountOfUser`) and determine whether a specific use has a valid ticket amount in the given `epoch`. And the internal helper routine `getTicketAmount()` is rather complex with numerous cases. If the whole set of users is rather huge with a long list of stale users, this routine will execute out of gas. A possible improvement will be limiting the number of iteration for each calculation by introducing an ID range in the calculation.

```
    function sumUptotalParticipant(uint256 epochNumber, uint256 endBlock) internal view
        returns(uint256){
        uint256 amountofParticipant;
        for(uint i = 1; i<amountOfUser; i++){
            uint256 registeredBlock = userDepositInfo[i].registeredDate;
            address userAddress = userDepositInfo[i].userAddress;
            if(registeredBlock <= endBlock){
                uint256 userTicketAmount = getTicketAmount(epochNumber,userAddress);
                if(userTicketAmount>0){
```

```
                    amountofParticipant += 1;
                }
            }
        }
        return amountofParticipant;
    }
```

Listing 3.2: `stETHPool::sumUptotalParticipant()`

Recommendation Revise the above routine to ensure it is executed properly without unwanted `OOG`.

Status The issue has been fixed by the following commit: `850421a`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Modulus` contract, there is a special account, `owner`, that plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that the `owner` account needs to be scrutinized. In the following, we use the `RoleRegistry` contract as an example and show the representative functions potentially affected by the privileges of the privileged account.

```
    function transferReserveAddr(address reserveAddr) external onlyOwner{
        _transferReserveAddr(reserveAddr);
    }
    function _transferReserveAddr(address reserveAddr) internal{
        reserveAddress = reserveAddr;
    }

    function setController(address newController) external onlyOwner{
        _transferControllerRole(newController);
    }

    function setrewardDistributor(address newDistributor) external onlyOwner{
        _transferRewardDistributorRole(newDistributor);
    }
    function _transferRewardDistributorRole(address newDistributor) internal{
        rewardDistributor = newDistributor;
    }
```

```
    function _transferControllerRole(address newController) internal{
        controller = newController;
    }

    function setRouter(address newRouter) external onlyOwner{
        _transferRouter(newRouter);
    }
```

Listing 3.3: Example Privileged Operations in `RoleRegistry`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged account 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 Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirm they are using a multi-sig account as the `owner`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Modulus` protocol, which is a decentralized platform for asymmetric yields distribution generated from staking. It offers the opportunity to earn significantly higher returns while keeping risk exposure controlled. This shift empowers participants to actively engage in a rewarding venture, transforming staking into a dynamic endeavor. Through innovative use of the `Chainlink Verified Random Function (VRF)`, the protocol selects 3 winners weekly based on deposit size and duration, ensuring an equitable opportunity for every participant to secure substantial rewards. The current code base is well organized and 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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