SMART CONTRACT AUDIT REPORT

for

TREVI PROTOCOL

Prepared By: Yiqun Chen

PeckShield

July 30, 2021

Document Properties

Client	Trevi
Title	Smart Contract Audit Report
Target	Trevi
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang, Shulin Bie, Xiaotao Wu
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 30, 2021	Xuxian Jiang	Final Release
1.0-rc	July 22, 2021	Xuxian Jiang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **Trevi** 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 several issues. This document outlines our audit results.

1.1 About Trevi

`Trevi` is an ERC20-based staking system with three main entities: `Archangel`, `Fountains`, and `Angels`. The `Archangel` records the relationship between `Fountains` and `Angels`, and manages the configuration of flash loan related functions. `Fountains` are the places where users may stake their assets in the corresponding fountain and receive the rewards from `Angels`, via which rewarders may manage their rewards schedule and configuration. The basic information of Trevi is as follows:

Table 1.1: Basic Information of Trevi

Item	Description
Name	Trevi
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 30, 2021

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

- https://github.com/dinngodev/trevi.git (b3f7fd3)

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

- https://github.com/dinngodev/trevi.git (6d68386)

1.2 About PeckShield

PeckShield Inc. [7] 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

	Likelihood		
Impact	*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 [6]:

- 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

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) [5], 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 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 Trevi implementation. 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 Trevi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely massUpdatePools During Pool Updates	Business Logic	Resolved
PVE-002	Informational	Managed Funds With Allowance To Fountain	Business Logic	Resolved
PVE-003	Low	Improved Sanity Checks For System Parameters	Coding Practices	Resolved

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 Timely massUpdatePools During Pool Updates

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: AngelBase
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The Trevi protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via add() and the weights of supported pools can be adjusted via set(). When analyzing the pool weight update routine set(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        IRewarder _rewarder,
        bool overwrite
    ) public onlyOwner {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint.to64();
        if (overwrite) {
            rewarder[_pid] = _rewarder;
        }
        emit LogSetPool(
            _pid,
            _allocPoint,
            overwrite ? _rewarder : rewarder[_pid],
            overwrite
        );
    }
```

Listing 3.1: `AngelBase::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated.

Status The issue has been fixed in the following commit: `3491f01`.

3.2 Managed Funds With Allowance To Fountain

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LendingPoolCollateralManager`, `LendingPool`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

Among all core functionalities provided in `Trevi`, `flashloan` is a disruptive one that allows users to borrow from the reserves within a single transaction, as long as the user returns the borrowed amount plus additional premium. In this section, we report an issue related to the `flashloan` feature.

```
    function flashLoan(
        IERC3156FlashBorrower receiver,
        address token,
        uint256 amount,
        bytes calldata data
    ) public override returns (bool) {
        uint256 fee = flashFee(token, amount);
        // send token to receiver
        lendingToken.safeTransfer(address(receiver), amount);
        require(
            receiver.onFlashLoan(msg.sender, token, amount, fee, data) ==
                _RETURN_VALUE,
            "ERC20FlashLoan: invalid return value"
        );
```

```
79         uint256 currentAllowance =
80             lendingToken.allowance(address(receiver), address(this));
81         uint256 totalDebt = amount.add(fee);
82         require(
83             currentAllowance >= totalDebt,
84             "ERC20FlashLoan: allowance does not allow refund"
85         );
86         // get token from receiver
87         lendingToken.safeTransferFrom(
88             address(receiver),
89             address(this),
90             totalDebt
91         );
92         address collector = flashLoanFeeCollector();
93         if (collector != address(0)) lendingToken.safeTransfer(collector, fee);
94 
95         return true;
96     }
```

Listing 3.2: ERC20FlashLoan::flashLoan()

To elaborate, we show above the code snippet of `flashLoan()` behind the feature. This particular routine implements the `flashloan` feature in a straightforward manner: It firstly transfers the funds to the specified receiver, then invokes the designated operation (`onFlashLoan()` - line 75), next transfers back the funds from the receiver or creates an equivalent borrow.

However, our analysis shows that the above logic may be abused to cause fund loss of an innocent user if the user previously specified certain allowances to the `Fountain` pool. Specifically, if a flashloan is launched by specifying the innocent user an the `receiver` argument, the `flashLoan()` execution follows the logic by firstly transferring the loan amount to `receiver`, invoking `onFlashLoan()` on the receiver, and then transferring the `totalDebt = amount.add(fee)` (no larger than the allowed spending amount) from the receiver back to the pool. Note that this flashloan is not initiated by the `receiver`, who unfortunately pays the premium associated with the flashloan.

Fortunately, this issue is largely mitigated with the return value check `require(receiver.onFlashLoan(msg.sender, token, amount, fee, data)== _RETURN_VALUE)` (lines 74−77). As far as the `receiver` does not return the expected value upon the `onFlashLoan()` call, the feature will not be abused.

Recommendation Revisit the design of the above routine in possibly avoiding initiating the `transferFrom()` call from the `Fountain` pool. However, considering the return value check, we believe there is no need to address it further.

Status The issue has been resolved.

3.3 Improved Sanity Checks For System Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [3]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Trevi protocol is no exception. Specifically, if we examine the AngelBase contract, they have defined a number of protocol-wide risk parameters, e.g., gracePerSecond, poolInfo, and lpToken. In the following, we show an example routine that allows for their changes.

```
    /// @notice Sets the grace per second to be distributed. Can only be called by the owner.
    /// @param _gracePerSecond The amount of GRACE to be distributed per second.
    function setGracePerSecond(uint256 _gracePerSecond) public onlyOwner {
        gracePerSecond = _gracePerSecond;
        emit LogGracePerSecond(_gracePerSecond);
    }
```

Listing 3.3: AngelBase::setGracePerSecond()

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        IRewarder _rewarder,
        bool overwrite
    ) public onlyOwner {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint.to64();
        if (overwrite) {
            rewarder[_pid] = _rewarder;
        }
        emit LogSetPool(
            _pid,
            _allocPoint,
            overwrite ? _rewarder : rewarder[_pid],
            overwrite
        );
    }
```

Listing 3.4: AngelBase::set()

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large `gracePerSecond` parameter will make every pool update operation reverted.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status The issue has been fixed in the following commit: `1d4afc4`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Trevi protocol. The system presents a unique offering as a trustless staking system to empower participating users to stake required funds to receive 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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