SMART CONTRACT AUDIT REPORT

for

PONTOON FINANCE

Prepared By: Yiqun Chen

PeckShield

October 2, 2021

Document Properties

Client	Pontoon Finance
Title	Smart Contract Audit Report
Target	Pontoon
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 2, 2021	Xuxian Jiang	Final Release
1.0-rc	July 6, 2021	Shulin Bie	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 Pontoon protocol, 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 Pontoon

Pontoon is a cross-chain liquidity mirror protocol. It provides a convenient one-click liquidity mirroring across ETH, BSC, HECO, xDAI, POLYGON, OPTIMISM with incentivized relayer network and liquidity mining for liquidity providers across the chains. It is a decentralized application based on smart contracts structure. With Pontoon, users can readily move coins between different blockchains.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Pontoon

Item	Description
Name	Pontoon
Website	https://pontoon.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 2, 2021

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

- https://gitlab.com/pontoonfi/pontoon.git (6a8df4f)

- https://gitlab.com/pontoonfi/pontoon-token.git (d767156)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://gitlab.com/pontoonfi/pontoon.git (803cf2a)
- https://gitlab.com/pontoonfi/pontoon-token.git (2fb6f3a)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 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) [9], 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

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Pontoon` 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 logic, 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	2	■■
Low	3	■■■
Informational	1	■
Total	6	

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 that 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 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Claimable Vesting Calculation	Business Logic	Resolved
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-003	Informational	Inconsistency Between Document and Implementation	Coding Practices	Resolved
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-005	Medium	Restricted Transferability of Pontoon-Pool Tokens	Business Logic	Resolved
PVE-006	Low	Incompatibility With Deflationary/Re-basing Tokens	Business Logic	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 Improved Claimable Vesting Calculation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PontoonTokenVesting`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `Pontoon` contract has a well-designed vesting schedule contract `PontoonTokenVesting`, which supports multiple rounds (e.g., `SEED`, `PRIVATE`, `STRATEGIC`, and `PUBLIC`) and multiple roles (e.g., `TEAM_AND_ADVISORS`, `COMMUNITY_REWARDS`, and `ECOSYSTEM_AND_MARKETING`). While examining the vesting logic, we notice the internal implementation can be improved.

To elaborate, we show below the related `_getTeamTokensAndVestingLeft()` routine. This routine is designed to compute the claimable vesting for the given team. Internally, there is a key variable `totalClaimableVesting` (line 555) that records the total number of claimable vesting. Note that this routine is invoked only when `block.timestamp >= _teamInfo.cliffEndTime`. And the total number of claimable vesting is currently computed as `((block.timestamp - _teamInfo.cliffEndTime)/ _teamInfo.vestingPeriod)+ 1`, which has an off-by-one issue and needs to be revised as `((block.timestamp - _teamInfo.cliffEndTime)/ _teamInfo.vestingPeriod)`.

```
551     function _getTeamTokensAndVestingLeft(TeamInfo memory _teamInfo) private view
            returns (uint256, uint256) {
552         // if sales is set in such a way after cliff time give all the tokens in one go
553         if (_teamInfo.vestingPeriod == 0) return _teamInfo.vestingsClaimed == 0 ? (
                _teamInfo.vestingTokens, 1) : (0, 0);
554
555         uint256 totalClaimableVesting = ((block.timestamp - _teamInfo.cliffEndTime) /
                _teamInfo.vestingPeriod) + 1;
556
557         uint256 claimableVestingLeft = totalClaimableVesting > _teamInfo.noOfVestings
558             ? _teamInfo.noOfVestings - _teamInfo.vestingsClaimed
```

```
            : totalClaimableVesting - _teamInfo.vestingsClaimed;

        uint256 unlockedTokens = _teamInfo.vestingTokens * claimableVestingLeft;

        return (unlockedTokens, claimableVestingLeft);
    }
```

Listing 3.1: `PontoonTokenVesting::_getTeamTokensAndVestingLeft()`

Note the same issue is also applicable to another helper routine `_getInvestorUnlockedTokensAndVestingLeft()`.

Recommendation Properly compute the claimable vesting for the given team information in the above `_getTeamTokensAndVestingLeft()`.

3.2 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PontoonToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
```

```
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `recoverToken()` routine in the `PontoonToken` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(token).transfer(destination, amount)` (line 57) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value). We may intend to replace it with `require(IERC20(token).safeTransfer(destination, amount), "Retrieve failed")`.

```
51     function recoverToken(
52         address token,
53         address destination,
54         uint256 amount
55     ) external onlyGovernance {
56         require(token != destination, "Invalid address");
57         require(IERC20(token).transfer(destination, amount), "Retrieve failed");
58         emit RecoverToken(token, destination, amount);
59     }
```

Listing 3.3: PontoonToken::recoverToken()

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`.

3.3 Inconsistency Between Document and Implementation

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PontoonTokenVesting`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

There is a misleading comment embedded among lines of solidity code, which brings unnecessary hurdles to understand and/or maintain the software. In particular, if we examine the various vesting schedules in the `PontoonTokenVesting` contract, the supply percentage allocated for the `ECOSYSTEM_AND_MARKETING` is 30.5%, not the commented 35.5% (line 96).

```
95     /**
96         supply : 35.5%
97         initial release : 10%
98         cliff: 0,
99         vesting schedule : linear vesting for 24 months months
100        vesting period : 1 (single vesting period is 1 sec)
101        no of vestings : sec in 24 months (as for each sec tokens will be released)
102     */
103    uint256 private constant ECOSYSTEM_AND_MARKETING_SUPPLY_PERCENT = 3050;
104    uint256 private constant ECOSYSTEM_AND_MARKETING_INITIAL_RELEASE_PERCENT = 1000;
105    uint256 private constant ECOSYSTEM_AND_MARKETING_CLIFF_PERIOD = 0;
106    uint256 private constant ECOSYSTEM_AND_MARKETING_VESTING_PERIOD = 1;
107    uint256 private constant ECOSYSTEM_AND_MARKETING_NO_OF_VESTINGS = 730 days;
```

Listing 3.4: `PontoonTokenVesting.sol`

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `Pontoon` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., adding new tokens and configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the untrusted `owner` account.

```
    function updateStartTime(uint256 _startAfter) external override onlyOwner() {
        require(_startAfter > 0, "Invalid startTime");
        require(block.timestamp < startTime, "Already started");

        uint256 _startTime = block.timestamp + _startAfter;

        _massUpdateCliffEndTime(_startTime);

        startTime = _startTime;
    }

    /**
     * @notice add, update or remove single investor
     * @param _amount for how much amount (in $) has investor invested. ex  100$ = 100 * 100 = 100,00
     * @dev to remove make amount 0 before it starts
     * @dev you can add, updated and remove any time
     */
    function addOrUpdateInvestor(
        RoundType _roundType,
        address _investor,
        uint256 _amount
    ) external override onlyOwner() {
        _addInvestor(_roundType, _investor, _amount);

        emit InvestorAdded(_roundType, _investor, _amount);
    }
```

Listing 3.5: A number of representative `setters` in `PontoonTokenVesting`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Note that a compromised `owner` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `Pontoon` design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed 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.

3.5 Restricted Transferability of PontoonPool Tokens

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PontoonPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

As a cross-chain liquidity mirror protocol, the `Pontoon` protocol allows for liquidity providers to add liquidity to the so-called `PontoonPool`. And the liquidity providers will get in return the corresponding share of the `PontoonPool` in terms of pool tokens. The `PontoonPool` pool tokens are `ERC20`-compliant tokens. While examining the token contract implementation, we notice the current transfer functionality is rather limited.

To elaborate, we show below the related code snippet of the `PontoonPool` contract. In particular, the `removeLiquidity()` function is designed to allow liquidity providers to remove their liquidity after the lockup period. This function has properly validated the available balance as well as the lockup period. However, as an `ERC20`-compliant token, the built-in functions `transfer()` and `transferFrom()` have not been enhanced to honor the lockup period. As a result, these built-in functions may not achieve the intended functionality.

```
    function removeLiquidity(uint256 _amount) external nonReentrant {
        require(
            liquidity[msg.sender].lpTokenBalance > 0,
            "Pool: sender is not a liquidity provider"
        );
        require(
            liquidity[msg.sender].unlockTime < block.timestamp,
            "Pool: Tokens are not yet available for withdrawal"
        );

        uint256 lpTokenAmount = _amount * (10**factor);

        require(
            liquidity[msg.sender].lpTokenBalance >= lpTokenAmount,
            "Pool: not enough token to remove"
        );
        // accrued fees are in the source token decimals whereas lp tokens is
        // always 18 decimal points. Thus we need to calculate lpFee with the source
            token's decimals precision
        uint256 lpFee = (accruedFee * lpTokenAmount) / totalSupply();

        // LPs are in 18 decimals whereas the source token does not always have 18.
        // We need to brign the figure down to the decimal points of the source tokens
        // since this is the liquidity that will be sent back to the sender
```

```
        uint256 withdrawAmount = _amount + lpFee;
        accruedFee -= lpFee;

        liquidity[msg.sender].lpTokenBalance -= lpTokenAmount;

        IERC20(token).safeTransfer(msg.sender, withdrawAmount);
        _burn(msg.sender, lpTokenAmount);
    }
```

Listing 3.6: `PontoonPool::removeLiquidity()`

Recommendation Enhance the built-in functions (e.g., `transfer()` and `transferFrom()` to properly honor the lockup period.

3.6 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PontoonPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In Pontoon, the `PontoonPool` contract is designed to be the main entry for interaction with liquidity-providing users. In particular, one entry routine, i.e., `deposit()`, accepts asset transfer-in and mints the corresponding pool tokens to represent the depositor's share in `PontoonPool`. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the pool. 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 addLiquidity(uint256 _amount) external {
        uint256 lpTokenAmount = _amount * (10**factor);

        liquidity[msg.sender].lpTokenBalance += lpTokenAmount;
        liquidity[msg.sender].unlockTime = block.timestamp + lockPeriod;

        IERC20(token).safeTransferFrom(msg.sender, address(this), _amount);
        _mint(msg.sender, lpTokenAmount);
    }
```

Listing 3.7: `PontoonPool::addLiquidity()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM/OHM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above `addLiquidity()` operation may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting 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 Pontoon for cross-chain transfers. In fact, Pontoon is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

4 Conclusion

In this audit, we have analyzed the `Pontoon` design and implementation. `Pontoon` is a cross-chain liquidity mirror protocol. It provides an interesting one-click liquidity mirroring across `ETH`, `BSC`, `HECO`, `xDAI`, `POLYGON`, `OPTIMISM` with incentivized relayer network and liquidity mining for liquidity providers across the chains. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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

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