SMART CONTRACT AUDIT REPORT

for

Increment Protocol

Prepared By: Xiaomi Huang

PeckShield

June 14, 2022

Document Properties

Client	Increment Finance
Title	Smart Contract Audit Report
Target	Increment Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 14, 2022	Xuxian Jiang	Final Release
1.0-rc1	June 12, 2022	Xuxian Jiang	Release Candidate #1

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 related smart contract source code of the `Increment` 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 Increment

USD-pegged stablecoins currently dominate the stablecoin supply in DeFi. This creates tremendous exchange rate risks for global non-USD participants. `Increment` builds global exchange rate products on `zkSync 2.0` to unleash the power of DeFi for citizens around the world. In particular, the protocol utilizes pooled virtual assets and `Curve V2`'s `CryptoSwap AMM` as the trading engine to enable multi-currency perpetual swaps such that DeFi users can hedge their USD exposure or speculate on global currency movements through on-chain perpetual swaps. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Increment Protocol

Item	Description
Name	Increment Finance
Website	https://increment.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 14, 2022

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

- https://github.com/Increment-Finance/increment-peckshield.git (8efaf7b)

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

- https://github.com/Increment-Finance/increment-peckshield.git (51b9712)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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.

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

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) [8], 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 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 implementation of the `Increment` 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	1	■
High	1	■
Medium	1	■
Low	2	■ ■
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 1 critical-severity vulnerability, 1 high-severity vulnerability, 1 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational suggestion.

Table 2.1: Key Increment Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Incorrect Withdrawal Logic in Vault::withdrawPartial()	Business Logic	Resolved
PVE-002	Low	Improved Insurance Logic in Insurance::removeInsurance()	Business Logic	Resolved
PVE-003	Low	Improved Validation Logic in ClearingHouse::setMinMargin()	Coding Practices	Resolved
PVE-004	High	Proper Trading Fee Settlement in _settleLpTradingFees()	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Informational	Removal of Redundant State And Code	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect Withdrawal Logic in Vault::withdrawPartial()

- ID: PVE-001
- Severity: Critical
- Likelihood: High
- Impact: High
- Target: `Vault`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `Increment`, there is a `Vault` contract that is designed to keep track of all token reserves for all markets. Accordingly, the `Vault` contract supports the interaction with users, including their deposits and withdraws. While examining the current withdrawal logic, we notice the current implementation needs to be improved.

To elaborate, we show below the implementation of the `withdrawPartial()` function. By design, the current logic allows the user to withdraw share of tokens from the user account across multi-collaterals. It comes to our attention that this function makes use of the following two local variables, including `amountToWithdraw` and `wadCollateralBalance`. The first one calculates the dollar-denominated withdrawal value while the second computes the collateral balance denominated in the collateral asset. However, these two variables are directly compared for the actual collateral withdrawal (line 144)! To fix, there is a need to use the same denomination before they can be used for comparision.

```
123     function withdrawPartial(
124         uint256 marketIdx,
125         address user,
126         uint256 reductionRatio,
127         bool isTrader
128     ) external override onlyClearingHouse {
129         if (reductionRatio > 1e18) revert Vault_WithdrawReductionRatioTooHigh();
130
131         // the amount to withdraw accross all collateral
132         int256 reserveValue = _getUserReserveValue(marketIdx, user, isTrader);
133         int256 amountToWithdraw = reserveValue.wadMul(reductionRatio.toInt256());
```

```

        Collateral[] memory collaterals = whiteListedCollaterals;
        int256 collateralBalance;
        int256 wadCollateralBalance;
        int256 tokenAmountToWithdraw;

        for (uint256 i = collaterals.length; i > 0; i--) {
            collateralBalance = isTrader ? traderBalances[user][marketIdx][i - 1] : lpBalances[user][marketIdx][i - 1];
            wadCollateralBalance = LibReserve.tokenToWad(collaterals[i - 1].decimals, collateralBalance);

            if (wadCollateralBalance >= amountToWithdraw) {
                tokenAmountToWithdraw = LibReserve.wadToToken(collaterals[i - 1].decimals, amountToWithdraw);

                withdraw(marketIdx, user, tokenAmountToWithdraw.toUint256(), collaterals[i - 1].asset, isTrader);
                break;
            } else {
                withdraw(marketIdx, user, collateralBalance.toUint256(), collaterals[i - 1].asset, isTrader);
                amountToWithdraw -= wadCollateralBalance;
            }
        }
    }
```

Listing 3.1: `Vault::withdrawPartial()`

Recommendation Revise the above logic to ensure `amountToWithdraw` and `wadCollateralBalance` are converted to the same denomination before their comparison.

Status This issue has been fixed in the following commit: `f9c37f8`.

3.2 Improved Insurance Logic in Insurance::removeInsurance()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Insurance`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Increment` protocol has a built-in `Insurance` contract that can be used to pay out `Vault` in case of default. While analyzing the logic behind the `Insurance` contract, we observe one of its functions (i.e., `removeInsurance()`) can be improved.

To elaborate, we show below the implementation of this `removeInsurance()` function. As the name indicates, this function is used to withdraw the remaining balance of the contract and by design can only be called by the owner of the contract. To ensure the removed amount is indeed part of the remaining balance, there is a validation check, i.e., `(lockedInsurance - amount)< tvl.wadMul (clearingHouse.insuranceRatio())` (line 88), which can be improved as follows: `(lockedInsurance <= amount)|| ((lockedInsurance - amount)< tvl.wadMul(clearingHouse.insuranceRatio()))`. In other words, to ensure it cannot withdraw more funds from insurance than the available funds, we suggest to add the explicit validation.

```
84     function removeInsurance(uint256 amount) external override onlyOwner {
85         // check insurance ratio after withdrawal
86         uint256 tvl = vault.getTotalValueLocked();
87         uint256 lockedInsurance = token.balanceOf(address(this));
88         if ((lockedInsurance - amount) < tvl.wadMul(clearingHouse.insuranceRatio()))
89             revert Insurance_InsufficientInsurance();
90 
91         // withdraw
92         emit InsuranceRemoved(amount);
93         IERC20Metadata(token).safeTransfer(msg.sender, amount);
94     }
```

Listing 3.2: `Insurance::removeInsurance()`

Recommendation Revise the above `removeInsurance()` to add the explicit check on `lockedInsurance >=amount`.

Status This issue has been fixed in the following commit: `4b8f123`.

3.3 Improved Validation Logic in ClearingHouse::setMinMargin()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ClearingHouse`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Increment` protocol is no exception. Specifically, if we examine the `ClearingHouse` contract, it has defined a number of protocol-wide risk parameters, such as `minMargin` and `minMarginAtCreation`. In the following, we show the corresponding routines that allow for their changes.

```
    function setMinMargin(int256 newMinMargin) external override onlyOwner {
        if (newMinMargin < 25e15) revert ClearingHouse_InsufficientMinMargin();
        if (newMinMargin > 3e17) revert ClearingHouse_ExcessiveMinMargin();

        minMargin = newMinMargin;
        emit MinMarginChanged(newMinMargin);
    }

    function setMinMarginAtCreation(int256 newMinMarginAtCreation) external override onlyOwner {
        if (newMinMarginAtCreation <= minMargin) revert ClearingHouse_InsufficientMinMargin();
        if (newMinMarginAtCreation > 5e17) revert ClearingHouse_ExcessiveMinMargin();

        minMarginAtCreation = newMinMarginAtCreation;
        emit MinMarginAtCreationChanged(newMinMarginAtCreation);
    }
```

Listing 3.3: ClearingHouse::setMinMargin() and ClearingHouse::setMinMarginAtCreation()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these 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 misconfiguration of `newMinMargin` may set it smaller than another related parameter `minMarginAtCreation`, hence violating the protocol design and potentially leading to unexpected execution outcome.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status This issue has been fixed in the following commit: `c1340e8`.

3.4 Proper Trading Fee Settlement in _settleLpTradingFees()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Perpetual`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

At the core of `Increment` is the `Perpetual` contract that implements the intended perpetual swap, which handles all the trading logic and interact with the `CryptoSwap` pool. While examining the liquidity-related operations, including addition or removal, we notice an issue in the current implementation.

In the following, we show below the related helper routine `_settleLpTradingFees()`, which is used to settle the trader's trading fee. It has a rather straightforward logic in calling another helper routine `_getLpTradingFees()` to compute the trader's fee in `tradingFeesEarned`. However, the current implementation fails to update the accounting on the trader's `totalTradingFeesGrowth`. In other words, there is a need to add the following statement before returning from the function: `lp.totalTradingFeesGrowth = global.totalTradingFeesGrowth`.

```
    function _settleLpTradingFees(
        LibPerpetual.LiquidityProviderPosition storage lp,
        LibPerpetual.GlobalPosition storage global
    ) internal view returns (uint256 tradingFeesEarned) {
        // settle lp trading fees
        tradingFeesEarned = _getLpTradingFees(lp, global);

        // reset lp.totalTradingFeesGrowth := trading fees index
        global.totalTradingFeesGrowth;

        return tradingFeesEarned;
    }
```

Listing 3.4: `Perpetual::_settleLpTradingFees()`

Recommendation Properly update the trader's `totalTradingFeesGrowth` to settle down the trader's fee.

Status This issue has been fixed in the following commit: `612226e`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In `Increment`, there is a privileged administrative account `owner`. This administrative account plays a critical role in governing and regulating the protocol-wide operations. It also has the privilege to control or govern the flow of assets within the protocol contracts. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `ClearingHouse` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
function setMinMarginAtCreation(int256 newMinMarginAtCreation) external override onlyOwner {
    if (newMinMarginAtCreation <= minMargin) revert ClearingHouse_InsufficientMinMargin();
    if (newMinMarginAtCreation > 5e17) revert ClearingHouse_ExcessiveMinMargin();

    minMarginAtCreation = newMinMarginAtCreation;
    emit MinMarginAtCreationChanged(newMinMarginAtCreation);
}

function setLiquidationReward(uint256 newLiquidationReward) external override onlyOwner {
    if (newLiquidationReward < 1e16) revert ClearingHouse_InsufficientLiquidationReward();
    if (newLiquidationReward >= minMargin.toUint256()) revert ClearingHouse_ExcessiveLiquidationReward();

    liquidationReward = newLiquidationReward;
    emit LiquidationRewardChanged(newLiquidationReward);
}

function setInsuranceRatio(uint256 newInsuranceRatio) external override onlyOwner {
    if (newInsuranceRatio < 1e17) revert ClearingHouse_InsufficientInsuranceRatio();
    if (newInsuranceRatio > 5e17) revert ClearingHouse_ExcessiveInsuranceRatio();

    insuranceRatio = newInsuranceRatio;
    emit InsuranceRatioChanged(newInsuranceRatio);
}
```

Listing 3.5: Example Privileged Operations in `ClearingHouse`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 This issue has been mitigated as the team confirms the use of a `multisig` or a `Governance` contract in charge of changing these risk parameters and executing these administrative functions.

3.6 Removal of Redundant State And Code

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The Increment protocol makes good use of a number of reference contracts, such as ERC20, SafeBEP20 , SafeMath, and Address, to facilitate its code implementation and organization. For example, the Perpetual smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the Insurance contract, the constructor function has a duplicate validation on the given _vault (lines 37 and 38). Also, the Perpetual contract has a public function provideLiquidity(), which updates the liquidity provider's lp.cumFundingRate twice (lines 367 and 373).

```
35     constructor(IERC20Metadata _token, IVault _vault) {
36         if (address(_token) == address(0)) revert Insurance_ZeroAddressConstructor(0);
37         if (address(_vault) == address(0)) revert Insurance_ZeroAddressConstructor(1);
38         if (address(_vault) == address(0)) revert Insurance_ZeroAddressConstructor(2);
39         token = _token;
40         vault = _vault;
41     }
```

Listing 3.6: Insurance::constructor()

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed by the following commits: 2d3a78f and f932606.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Increment protocol, which builds global exchange rate products on zkSync 2.0 to unleash the power of DeFi for citizens around the world. In particular, the protocol utilizes pooled virtual assets and Curve V2's CryptoSwap AMM as the trading engine to enable multi-currency perpetual swaps such that DeFi users can hedge their USD exposure or speculate on global currency movements through on-chain perpetual swaps. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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