SMART CONTRACT AUDIT REPORT

for

Streams

Prepared By: Xiaomi Huang

PeckShield

June 10, 2022

Document Properties

Client	Streams
Title	Smart Contract Audit Report
Target	Streams
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 10, 2022	Xuxian Jiang	Final Release
1.0-rc	May 8, 2022	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 related smart contract source code of the Streams smart contracts, 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 audited 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 Streams

The Streams smart contracts are batchers that batch multiple user transactions into one, interact with other DeFi protocols and return the tokens back to the users. For example, they are interacting with the Idle Finance protocol on both Polygon and Ethereum chains. The Idle Finance is a decentralized protocol that algorithmically optimizes digital asset allocations across leading DeFi protocols to maximize yield or balance the risk/return ratio. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Streams

Item	Description
Name	Streams
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 10, 2022

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

- https://github.com/StreamsXYZ/smart-contracts.git (9222f98)

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

- https://github.com/StreamsXYZ/smart-contracts.git (7e07ff2)

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

Likelihood

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

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

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) [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 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 `Streams` 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	1	■
Low	4	■ ■ ■ ■
Informational	0	
Total	5	

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 medium-severity vulnerability and 4 low-severity vulnerabilities.

Table 2.1: Key Streams Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-002	Low	Improved Validation in DepositBatcher::mint()	Coding Practices	Resolved
PVE-003	Low	Proper RedeemedData Bookkeeping in WithdrawRouter	Business Logic	Resolved
PVE-004	Low	Revised Protocol Management in Factory	Business Logic	Resolved
PVE-005	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 contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

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

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 stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the IERC20 interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of USDT's `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT::transfer()

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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In current implementation, if we examine the `FeeHandler::processFees()` routine that is designed to process the fees. To accommodate the specific idiosyncrasy, there is a need to user `safeTransferFrom()`, instead of `transferFrom()` (line 59).

```
49    function processFees(
50        address user,
51        address[] memory token,
52        uint256[] memory amount
53    ) external override onlyRole(GOVERNOR_ROLE) {
54        for (uint256 i = 0; i < token.length; i++) {
55            require(
56                IERC20(token[i]).allowance(user, address(this)) >= amount[i],
57                Errors.VL_INSUFFICIENT_ALLOWANCE
58            );
59            IERC20(token[i]).transferFrom(user, treasury, amount[i]);
60            feesPaid[user][token[i]] += amount[i];
61            emit FeeClaimed(user, token[i], amount[i]);
62        }
63    }
```

Listing 3.2: `FeeHandler::processFees()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by the following commits: `e8cd981` and `4b10946`.

3.2 Improved Validation in DepositBatcher::mint()

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

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Streams` is no exception. Specifically, if we examine the `DepositBatcher` contract, it has defined a number of protocol-wide risk parameters, such as `totalPerTenure` and `tenures`.

```
    function mint(
        bytes32[] memory protocols,
        uint256[] memory amounts,
        uint256 total
    ) external virtual override onlyWhitelisted nonReentrant returns (bool) {
        /// @dev destructures the user information.
        address user = _msgSender();
        bool result = true;

        /// @dev validates & transfers usdc from user to smart contract.
        require(_usdc.balanceOf(user) >= total, Errors.VL_INSUFFICIENT_BALANCE);
        require(
            _usdc.allowance(user, address(this)) >= total,
            Errors.VL_INSUFFICIENT_ALLOWANCE
        );
        result = result && _usdc.transferFrom(user, address(this), total);

        for (uint256 i = 0; i < protocols.length; i++) {
            address _protocolAddress = _factory.fetchProtocolAddressL2(
                protocols[i]
            );
            if (_protocolAddress != address(0)) {
                result = result && instantMint(
                    user,
                    protocols[i],
                    amounts[i],
                    _protocolAddress
                );
            } else {
                result = result && batch(user, protocols[i], amounts[i]);
            }
        }

        return result;
    }
```

Listing 3.3: DepositBatcher::mint()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. For example, if we examine the above `mint()` function, it can be improved by validating the input arrays of `protocols` and `amounts` share the same length. Moreover, the given `total` is indeed equal to the sum of elements in the `amounts` array.

The same issue is also applicable to other routines, including `DepositBatcher::_processMessageFromRoot()` and `WithdrawBatcher::redeem()/_processBatch()`.

Recommendation Validate the given input arguments to ensure they fall in an appropriate range.

Status The issue has been fixed by the following commits: `9820bd6`, `288bf02`, `9224c13`, and `e0344ec`.

3.3 Proper RedeemedData Bookkeeping in WithdrawRouter

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: WithdrawRouter
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Streams smart contracts have a WithdrawRouter that is designed to interact with protocol children and redeem the tokens. While reviewing its logic, we notice one specific function that is used to process messages from protocol children can be improved.

Specifically, we show below the related _processMessageFromChild() function. This function implements a rather straightforward logic in handling the messages from protocol children. It comes to our attention that the variable tunneldata has three fields: batchId, protocols, and amounts. The current handler only keeps track of the first and last fields, but not the middle one!

```
    function _processMessageFromChild(bytes memory message)
        internal
        virtual
        override
    {
        RedemptionData memory data = abi.decode(message, (RedemptionData));
        RedeemedData memory tunneldata;

        tunneldata.batchId = data.batchId;
        uint256[] memory amounts = new uint256[](data.protocols.length);

        bool result = true;
        for (uint256 i = 0; i < data.protocols.length; i++) {
            address protocolAddress = factory
                .fetchProtocolInfo(data.protocols[i])
                .protocolAddressL1;
            address tokenAddress = factory
                .fetchProtocolInfo(data.protocols[i])
                .tokenAddressL1;
            IERC20(tokenAddress).transfer(protocolAddress, data.amounts[i]);
            (result, amounts[i]) = IProtocolL1(protocolAddress).redeemProtocolToken(
                data.amounts[i]
            );
        }

        tunneldata.amounts = amounts;
        redemptionStateInfo[data.batchId] = abi.encode(tunneldata);
        // _sendMessageToChild(abi.encode(tunneldata));
```

```
    }
```

Listing 3.4: `WithdrawRouter::_processMessageFromChild()`

Recommendation Properly keep track of the full `tunnelData` in the storage state `redemptionStateInfo`
.

Status The issue has been fixed by this commit: `de1e367`.

3.4 Revised Protocol Management in Factory

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Factory`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Streams` smart contracts have a `Factory` contract that is used to store all the record of protocol children contracts. In particular, it contains two main privileged functions `addProtocol()` and `removeProtocol()`: the first one adds a new protocol children contract while the second removes an existing one.

Our analysis shows that the first `addProtocol()` function can be improved by validating the given protocol children contract for addition is indeed a new one. Similarly, the second `removeProtocol()` function can be improved by validating the given protocol child contract for removal is indeed an existing one.

```
    function addProtocol(
        address protocolAddressL1,
        address protocolAddressL2,
        address tokenAddressL1,
        address tokenAddressL2,
        address stablecoinL1,
        address stablecoinL2,
        bytes32 protocolName
    ) external virtual override onlyRole(GOVERNOR_ROLE) returns (bool) {
        FactoryData memory tunneldata = FactoryData(
            protocolName,
            tokenAddressL1,
            tokenAddressL2,
            protocolAddressL1,
            protocolAddressL2,
            stablecoinL1,
            stablecoinL2
```

```
        );

        protocol[protocolName] = tunneldata;

        _length += 1;
        /// uncomment during mainnet deployment.
        // _sendMessageToChild(abi.encode(tunneldata));
        emit ProtocolAdded(
            protocolName,
            protocolAddressL1,
            protocolAddressL2,
            tokenAddressL1,
            tokenAddressL2,
            stablecoinL1,
            stablecoinL2
        );
        return true;
    }

    /// @dev refer {IFactory-removeProtocol}
    function removeProtocol(bytes32 protocolName)
        external
        virtual
        override
        onlyRole(GOVERNOR_ROLE)
        returns (bool)
    {
        FactoryData memory tunneldata = FactoryData(
            protocolName,
            address(0),
            address(0),
            address(0),
            address(0),
            address(0),
            address(0)
        );

        protocol[protocolName] = tunneldata;

        _length -= 1;
        /// uncomment during mainnet deployment.
        // _sendMessageToChild(abi.encode(tunneldata));
        emit ProtocolDeleted(protocolName);

        return true;
    }
```

Listing 3.5: Factory::addProtocol()/removeProtocol()

Recommendation Validate the protocol children contract for its proper addition and removal.

Status The issue has been fixed by this commit: `01ad375`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In `Streams`, there are special administrative accounts (with `GOVERNOR_ROLE`, `DEFAULT_ADMIN_ROLE`, or `owner`). These accounts play a critical role in governing and regulating the contract-wide operations (e.g., configure parameters and execute privileged operations). They also have the privilege to control or govern the flow of assets managed by the smart contracts. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine their related privileged accesses in current smart contracts.

```
    function execute() external virtual override onlyRole(GOVERNOR_ROLE) returns (bool)
        {
        uint256 batchId = _currentBatch;
        WithdrawBatch storage b = _batch[_currentBatch];
        /// @dev does the sanitary check to make sure the batchId is valid
        require(b.status == BatchStatus.LIVE, Errors.VL_BATCH_NOT_ELLIGIBLE);

        RedemptionData memory tunneldata;

        uint256[] memory amounts = new uint256[](b.protocols.length);

        /// @dev constructs an array to be sent via data tunnel.
        for (uint256 i = 0; i < b.protocols.length; i++) {
            amounts[i] = b.tokens[b.protocols[i]];
            IERC20L2(_factory.fetchTokenAddressL2(b.protocols[i])).withdraw(
                b.tokens[b.protocols[i]]
            );
        }

        /// @dev constructs the tunnel data.
        tunneldata.batchId = batchId;
        tunneldata.protocols = b.protocols;
        tunneldata.amounts = amounts;

        b.status = BatchStatus.BATCHED;
        _currentBatch += 1;

        /// @dev send the BatchData via Data Tunnel.
        _sendMessageToRoot(abi.encode(tunneldata));
        emit UpdateBatch(batchId, BatchStatus.BATCHED);
        return true;
```

```
    }

    /// @dev in case of auto distribution fails.
    /// Note: Internal function calls during matic state tunnel update might fail.
    function manualProcessMessageFromRoot(
        uint256 batchId,
        bytes[] memory swapExtraData
    ) external onlyRole(GOVERNOR_ROLE) nonReentrant {
        RedeemedData memory data = abi.decode(receivedStateInfo[batchId], (RedeemedData)
            );
        _processBatch(abi.encode(data), swapExtraData);
    }
```

Listing 3.6: Example Privileged Operations in `WithdrawBatcher`

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

Recommendation Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And 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 confirmed by the team. The team clarifies that a multi-sig account is used as the admin wallet.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Streams`, which are batchers that batch multiple user transactions into one, interact with other DeFi protocols and return the tokens back to the users. For example, they are interacting with the `Idle Finance` protocol on both `Polygon` and `Ethereum` chains to optimize digital asset allocations across leading `DeFi` protocols to maximize yield or balance the risk/return ratio. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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.