SMART CONTRACT AUDIT REPORT

for

Stackit Protocol

Prepared By: Xiaomi Huang

PeckShield
February 25, 2023

Document Properties

Client	Stackit Protocol
Title	Smart Contract Audit Report
Target	Stackit
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 25, 2023	Xuxian Jiang	Final Release
1.0-rc	February 18, 2023	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 source code of the `Stackit` 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 Stackit

`Stackit` protocol is designed to setup an automatic `dollar cost averaging (DCA)` strategy in seconds. The strategy is the practice of systematically investing equal amounts of money at regular intervals, regardless of the price of a security. The protocol provides an easy and user-friendly interface to users with less exposure to `DeFi`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Stackit

Item	Description
Name	Stackit Protocol
Website	https://stackit.finance/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 25, 2023

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

- https://github.com/ArchonSpear/stackit-audit.git (dc76f1a)

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

- https://github.com/ArchonSpear/stackit-audit.git (bc3f413)

1.2 About PeckShield

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

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be 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

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

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. 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 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 design and implementation of the `Stackit` protocol smart contracts. 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	2	■■
Low	2	■■
Informational	1	■
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 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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-002	Informational	Suggested Event Generations on Setting Changes	Coding Practices	Confirmed
PVE-003	Low	Incorrect Length Enforcement in _createStreamMultiple()	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Inconsistent Reentrancy Enforcement in Stackit	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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: N/A

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 transferFrom() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transferFrom() interface with a bool return value. As a result, the call to transferFrom() may expect a return value. With the lack of return value of USDT's transferFrom(), the call will be unfortunately reverted.

```
    // Forward ERC20 methods to upgraded contract if this one is deprecated
    function transferFrom(address _from, address _to, uint _value) public whenNotPaused
        {
        require(!isBlackListed[_from]);
        if (deprecated) {
            return UpgradedStandardToken(upgradedAddress).transferFromByLegacy(msg.
                sender, _from, _to, _value);
        } else {
            return super.transferFrom(_from, _to, _value);
        }
    }
```

Listing 3.1: USDT::transferFrom()

Because of that, a normal call to `transferFrom()` is suggested to use the safe version, i.e., `safeTransferFrom()`, 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.

In current implementation, if we examine the `StackitV3::activate()` routine that is designed to activate the `DCA` stream. To accommodate the specific idiosyncrasy, there is a need to user `safeTransferFrom()`, instead of `transferFrom()` (line 455).

```
35     function activate(
36         uint256 _amount,
37         uint256 _intervalInSec,
38         address _buyWith,
39         uint256 _refLink,
40         address _toBuy,
41         uint256 _iteration,
42         uint256 _startTime,
43         bool _yieldActive)
44         external whenNotPaused {
45         address buyWith = _buyWith;
46         require(_amount >= minimumAmountPerBuy[buyWith] ,"Please add more than the
               minimum amount per buy");
47         if (msg.sender != owner()) {
48             require(_startTime > block.timestamp,"Stream start date should be in the
                   future");
49         }
50         require(_iteration > 1,"Minimum two iterations required");

52         if (assetHasYield[buyWith] == false && _yieldActive) {
53             revert("Asset does not have yield, please change selection");
54         }

56         if (!IReferral(Referrals).isUserSignedUp(msg.sender)) {
57             if (_refLink != 0) {
58                 IReferral(Referrals).signUpWithReferral(_refLink,msg.sender);
59             } else {
60                 IReferral(Referrals).signUp(msg.sender);
61             }
62         }

64         // in any case, push new struct to records
65         uint256 count = recordcounter.length;
66         streamYield[count] = _yieldActive;

68         Stream storage s = streams[count];
69         StreamAmounts storage sa = streamAmounts[count];

71         //we compute the total amount that will be spent
72         uint256 totalAmount = _iteration.mul(_amount);
73         uint256 totalAmountAdjusted = totalAmount;
```

```
        recordcounter.push(RecordCounter(count));
        //create the stream
        s.amount = _amount;
        s.totalAmount = totalAmount;
        s.interval = _intervalInSec;
        s.startTime = _startTime;
        s.lastSwap = 0;
        s.isactive = 1;
        s.buyWithSwapped = 0;
        s.toBuyReceived = 0;
        s.buyWith = buyWith;
        s.toBuy = _toBuy;
        s.owner = msg.sender;
        s.iteration = _iteration;
        s.shares = 0;

        amountLeftInStream[count] = totalAmountAdjusted;

        list[msg.sender].push(count);
        //deposit to our contracts
        IERC20(buyWith).transferFrom(msg.sender, address(this), totalAmountAdjusted);

        if (streamYield[count]) {
            _depositToVault(count, totalAmountAdjusted, false);
        } else {
            sa.assetAmount = sa.assetAmount.add(totalAmountAdjusted);
        }
        allStreams.push(count);
        streamNature[count] = false;
    }
```

Listing 3.2: `StackitV3::activate()`

In the meantime, we also suggest to use the safe-version of `approve()/transfer()/transferFrom()` in other related routines, including `activate()`, `activateMultiple()`, `topUp()`, and `withdrawAsset()`.

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

Status This issue has been fixed in the following commit: `9ccadef`.

3.2 Suggested Event Generations on Setting Changes

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

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `StackitV3` contract as an example. This contract has public privileged functions that are used to configure important parameters. While examining the events that reflect their changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `oracle` is being updated in `setOracle`, there is no respective event being emitted to reflect the update of `DIADaptor` (line 344).

```
334     function setAssetYield(address _asset, bool _yield) public onlyOwner {
335         assetHasYield[_asset] = _yield;
336     }

338     function setReferralFeesAggregator(address _ReferralFeesAggregator) public onlyOwner
             {
339         ReferralFeesAggregator = _ReferralFeesAggregator;
340     }


343     function setOracle(address _DIADaptor) public onlyOwner {
344         DIADaptor = _DIADaptor;
345     }

347     function setStargateParams(address _asset, uint16 _poolId, address _stargateAsset)
            public onlyOwner {
348         stargatePoolID[_asset] = _poolId;
349         stargateAsset[_asset] = _stargateAsset;
350     }

352     function setStargateRouter(address _router) public onlyOwner {
353         stargateRouter = _router;
354     }
```

Listing 3.3: Example Setters in `StackitV3`

Recommendation Properly emit respective events when important parameters become effective.

Status This issue has been confirmed.

3.3 Incorrect Length Enforcement in _createStreamMultiple()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `StackitV3`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Stackit` protocol allows for the activation of `DCA` streams (via the `activate()` and `activateMultiple()` functions). While examining the `activateMultiple()` function, we notice it needs to validate the input arguments. Our analysis shows that the current validation is incomplete.

Specifically, we show below the implementation of the related helper routine `_createStreamMultiple()` inside the `activateMultiple()` method. This helper routine performs the same-length validation between the user-provided input, i.e., `assetLen` and `assetLen`. However, the current implementation compares `assetLen` with itself (line 545), which always yields true!

```
540    function _createStreamMultiple(uint256 count, address[] memory _streamBasketOfAsset,
           uint256[] memory _streamAggregateRepartition) internal {
541            streamNature[count] = true;
542            streamBasketOfAsset[count] = _streamBasketOfAsset;
543            uint256 assetLen = _streamBasketOfAsset.length;
544            uint256 reapLen = _streamAggregateRepartition.length;
545            require(assetLen == assetLen, "Invalid repatition among assets");
546
547            uint256 mathCheck;
548            for (uint256 i = 0; i < assetLen; i++) {
549                uint256 percent = _streamAggregateRepartition[i];
550                require(percent >= 10, "Allocation cannot be less than 10%");
551                aggregateRepartitionForStream[count][_streamBasketOfAsset[i]] = percent;
552                mathCheck = mathCheck.add(_streamAggregateRepartition[i]);
553            }
554        require(mathCheck == 100, "Uneven percentage repartition among assets");
555    }
```

Listing 3.4: `StackitV3::_createStreamMultiple()`

Recommendation Properly validate the user input in the above `_createStreamMultiple()` helper routine.

Status This issue has been fixed in the following commit: `9ccadef`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Stackit` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and treasury adjustment). It also has the privilege to regulate or govern the flow of assets within the protocol.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Stackit` protocol.

```
    function setKeeper(address _keeper) public onlyOwner {
        require(_keeper != address(0), "No ded address ser");
        Keeper = _keeper;
    }

    function setAggregator(address _AGGREGATION_ROUTER_V5) public onlyOwner {
        AGGREGATION_ROUTER_V5 = _AGGREGATION_ROUTER_V5;
    }

    function setAssetDecimals(address _asset, uint256 _decimals) public onlyOwner {
        assetDecimals[_asset] = _decimals;
    }

    function setAssetYield(address _asset, bool _yield) public onlyOwner {
        assetHasYield[_asset] = _yield;
    }

    function setReferralFeesAggregator(address _ReferralFeesAggregator) public onlyOwner
     {
        ReferralFeesAggregator = _ReferralFeesAggregator;
    }


    function setOracle(address _DIADaptor) public onlyOwner {
        DIADaptor = _DIADaptor;
    }
```

```
    function setStargateParams(address _asset, uint16 _poolId, address _stargateAsset)
        public onlyOwner {
        stargatePoolID[_asset] = _poolId;
        stargateAsset[_asset] = _stargateAsset;
    }

    function setStargateRouter(address _router) public onlyOwner {
        stargateRouter = _router;
    }

    function setTreasury(address _treasury) public onlyOwner {
        treasury = _treasury;
    }

    function setReferrals(address _Referrals) public onlyOwner {
        Referrals = _Referrals;
    }

    function setFees(
        uint256 _treasuryInboundFees,
        uint256 _treasuryOutboundFees,
        uint256 _trxCostPercentFee
        ) public onlyOwner {
        treasuryInboundFees = _treasuryInboundFees;
        treasuryOutboundFees = _treasuryOutboundFees;
        trxCostPercentFee = _trxCostPercentFee;
    }

    function addAssetVault(
        address _asset,
        address _vault) public onlyOwner {
        Vaults storage v = vaults[_asset];
        v.asset = _asset;
        v.vault = _vault;
    }

    function setMinimumAmount(uint256 _buyAmount, address _asset) public onlyOwner {
        require(_buyAmount > 0, "No Zeroguerino");
        minimumAmountPerBuy[_asset] = _buyAmount;
    }
```

Listing 3.5: Various Privileged Operations in Stackit

We emphasize that the privilege assignment with various protocol contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure.

We point out that a compromised `owner` account would allow the attacker to invoke the above `drainTo` to steal funds in current protocol, which directly undermines the assumption of the `AirSwap` protocol.

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.

Status This issue has been confirmed and partially mitigated with a multi-sig account to regulate the governance privileges.

3.5 Inconsistent Reentrancy Enforcement in Stackit

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StackitV3
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [11] exploit, and the recent Uniswap/Lendf.Me hack [10].

We notice there are occasions where the re-entrancy avoidance is not consistently enforced. Inside the StackitV3 contract, a number of functions are protected with the nonReentrant modifier. However, there are still a number of other functions that are not protected with the nonReentrant modifier. Specifically, we show below the activateMultiple() routine without the nonReentrant modifier. The same issue is also applicable to other routines, including activate() and executeBuy().

```
    function activateMultiple(
        uint256 _amount,
        uint256 _intervalInSec,
        address _buyWith,
        uint256 _refLink,
        uint256 _iteration,
        address[] memory _streamBasketOfAsset,
        uint256[] memory _streamAggregateRepartition,
        uint256 _startTime,
        bool _yieldActive
        )
```

```
        external whenNotPaused {
        address buyWith = _buyWith;
        require(_amount >= minimumAmountPerBuy[buyWith] ,"Please add more than the minimum amount per buy");
        if (msg.sender != owner()) {
            require(_startTime > block.timestamp,"Stream start date should be in the future");
        }
        require(_iteration > 1,"Minimum two iterations required");
        require(_streamBasketOfAsset.length != 0, "Must be a multiple stream");

        if (assetHasYield[buyWith] == false && _yieldActive) {
            revert("Asset does not have yield, please change selection");
        }

        if (!IReferral(Referrals).isUserSignedUp(msg.sender)) {
            if (_refLink != 0) {
                IReferral(Referrals).signUpWithReferral(_refLink,msg.sender);
            } else {
                IReferral(Referrals).signUp(msg.sender);
            }
        }

        // in any case, push new struct to records
        uint256 count = recordcounter.length;
        streamYield[count] = _yieldActive;

        Stream storage s = streams[count];
        StreamAmounts storage sa = streamAmounts[count];

        //we compute the total amount that will be spent
        uint256 totalAmount = _iteration.mul(_amount);
        uint256 totalAmountAdjusted = totalAmount;

        recordcounter.push(RecordCounter(count));

        s.amount = _amount;
        s.totalAmount = totalAmount;
        s.interval = _intervalInSec;
        s.startTime = _startTime;
        s.lastSwap = 0;
        s.isactive = 1;
        s.buyWithSwapped = 0;
        s.toBuyReceived = 0;
        s.buyWith = buyWith;
        s.owner = msg.sender;
        s.iteration = _iteration;
        s.shares = 0;

        amountLeftInStream[count] = totalAmountAdjusted;
        _createStreamMultiple(count,_streamBasketOfAsset,_streamAggregateRepartition);
```

```
        list[msg.sender].push(count);
        //deposit to our contracts
        IERC20(buyWith).transferFrom(msg.sender, address(this), totalAmountAdjusted);
        if (streamYield[count]) {
            _depositToVault(count, totalAmountAdjusted, false);
        } else {
            sa.assetAmount = sa.assetAmount.add(totalAmountAdjusted);
        }
        allStreams.push(count);
        streamNature[count] = true;
    }
```

Listing 3.6: `StackitV3::activateMultiple()`

Recommendation Consistently enforce the `nonReentrant` modifier in public functions.

Status This issue has been fixed in the following commit: `9ccadef`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Stackit` protocol, which is designed to setup an automatic `dollar cost averaging (DCA)` strategy in seconds. The strategy is the practice of systematically investing equal amounts of money at regular intervals, regardless of the price of a security. The protocol provides an easy and user-friendly interface to users with less exposure to `DeFi`. 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-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.

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.