SMART CONTRACT AUDIT REPORT

for

SYMBLOX

Prepared By: Shuxiao Wang

Hangzhou, China

October 15, 2020

Document Properties

Client	Symblox
Title	Smart Contract Audit Report
Target	Yield Farming
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 15, 2020	Xuxian Jiang	Final Release
0.2	October 12, 2020	Xuxian Jiang	Additional Findings
0.1	October 10, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of `Symblox Yield Farming`, we in the report outline 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 Symblox

`Symblox` is a cross-chain synthetic asset issuance and derivatives trading protocol. Users can stake any token asset that is supported by the protocol as collateral. Once staked, users can mint synthetic assets that can then be traded on the protocol's exchange. In turn, the protocol rewards users with its governance token `SYX`. The audited `Symblox Yield Farming` is an effective trendy way to build up the liquidity pool to be used as collaterals for synthetic assets for the next phase of adoption.

The basic information of the `Symblox` protocol is as follows:

Table 1.1: Basic Information of `Symblox`

Item	Description
Issuer	Symblox
Website	https://symblox.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 15, 2020

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

- https://github.com/symblox/symblox-yield-farming (76c40ce)

1.2 About PeckShield

PeckShield Inc. [18] 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 [12]:

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 `Symblox Yield Farming`. 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	1	■
Medium	0	
Low	6	■■■■■■
Informational	1	■
Total	8	

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 high-severity vulnerability, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of The Bam Finance Protocol

ID	Severity	Title	Category	Status
PVE-001	High	Potential Denial-Of-Service in deposit()/withdraw()	Business Logics	Fixed
PVE-002	Low	Missed initializer Modifier in BaseConnector::initialize()	Coding Practices	Fixed
PVE-003	Low	Duplicate Pool Detection and Prevention	Business Logics	Fixed
PVE-004	Informational	Recommended Explicit Pool Validity Checks	Security Features	Fixed
PVE-005	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logics	Confirmed
PVE-006	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Fixed
PVE-007	Low	Improved Logic in getMultiplier()	Business Logics	Fixed
PVE-008	Low	Lack of Native Token Support in gulp()	Business Logics	Fixed

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 Potential Denial-Of-Service in deposit()/withdraw()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact:High
- Target: `WvlxConnector`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

The `Symblox` protocol is influenced by `SushiSwap` and shares several common components, i.e., `staking pools`, `reward manager`, and `reward token`. In addition, `Symblox` provides a `connector` that is specific for a user and the connecting pool. The `connector` provides a wrapper to facilitate users to perform staking and unstaking.

While analyzing the `connector` implementation, we notice possible denial-of-service issues in both `deposit()` and `withdraw()` routines in the `WvlxConnector` contract. To elaborate, we first show below the `deposit()` routine.

```
    function deposit(uint256)
        external
        payable
        onlyOwner
        returns (uint256 wvlxAmount)
    {
        // Cast lpToken from address to address payable
        address payable recipient = address(uint160(address(lpToken)));
        // Send to wrap VLX contract
        recipient.sendValue(msg.value);
        // Make sure the amount received is the same as the one sent
        wvlxAmount = IERC20(lpToken).balanceOf(address(this));
        require(wvlxAmount == msg.value, "ERR_WVLX_RECEIVED");
        //
        // Deposit to the RewardManager
        //
```

```
36          stakeLpToken(wvlxAmount);
37
38          emit LogDeposit(msg.sender, msg.value);
39      }
```

Listing 3.1: WvlxConnector.sol

The `deposit()` routine works by wrapping the native token into `WVLX` (lines 29 – 31) and then depositing the wrapped `WVLX` at the same amount into the staking pool (line 36). However, the requirement at line 32 restricts the staked amount is the same as the transferred value of the native token. While reasonable, it may introduce a denial-of-service attack if a malicious actor may intentionally transfer a tiny amount of native tokens into the `WvlxConnector` contract. By doing so, the user will not be able to stake successfully.

The `withdraw()` routine shares a similar issue. To elaborate, we show its code snippet below. The sanity check at line 56 will prevent the staked assets from being withdrawn. In other words, the staked funds will be locked in the contract forever.

```
41      function withdraw(uint256 amount, uint256)
42          external
43          onlyOwner
44          returns (uint256 tokenAmountOut)
45      {
46          //
47          // Withdraw the liquidity pool tokens from RewardManager
48          //
49          unstakeLpToken(amount);
50
51          //
52          // Withdraw VLX from wvlx (lptoken)
53          //
54          IWvlx(lpToken).withdraw(amount);
55
56          require(address(this).balance == amount, "ERR_VLX_RECEIVED");
57
58          tokenAmountOut = address(this).balance;
59          msg.sender.transfer(tokenAmountOut);
60
61          emit LogWithdrawal(msg.sender, tokenAmountOut);
62      }
```

Listing 3.2: WvlxConnector.sol

Note that the `BptConnector` contract is safe and does not share this issue.

Recommendation Revise the above-mentioned requirements to thwart possible denial-of-service attacks. An example revision of the `withdraw()` routine is shown below.

```
    function withdraw(uint256 amount, uint256)
        external
        onlyOwner
```

```
        returns (uint256 tokenAmountOut)
    {
        //
        // Withdraw the liquidity pool tokens from RewardManager
        //
        unstakeLpToken(amount);

        //
        // Withdraw VLX from wvlx (lptoken)
        //
        IWvlx(lpToken).withdraw(amount);

        require(address(this).balance >= amount, "ERR_VLX_RECEIVED");

        tokenAmountOut = address(this).balance;
        msg.sender.transfer(tokenAmountOut);

        emit LogWithdrawal(msg.sender, tokenAmountOut);
    }
```

Listing 3.3: WvlxConnector.sol

Status The issue has been fixed by this commit: `b06fb64cb3856aab5a1da05c032143c99db829cd`.

3.2 Missed initializer Modifier in BaseConnector::initialize()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseConnector`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1041 [2]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

However, a follow-up caveat is that during a contract's lifetime, its constructor is guaranteed to be called exactly once (and it happens at the very moment of being deployed). But a regular function may be called multiple times! In order to ensure that a contract will only be initialized once, we need to guarantee that the chosen `initialize()` function can be called only once during the entire lifetime. This guarantee is typically implemented as a modifier named `initializer`.

```
    function initialize(
        address _owner,
        address _rewardManager,
        address _lpToken,
        uint8 _rewardPoolId
    ) external {
        require(_owner != address(0), "ERR_OWNER_INVALID");
        require(_rewardManager != address(0), "ERR_REWARD_MANAGER");
        require(_lpToken != address(0), "ERR_LP_TOKEN");

        syxOwnable.initialize(_owner);
        rewardManager = IRewardManager(_rewardManager);
        lpToken = _lpToken;
        rewardPoolId = _rewardPoolId;

        emit LogInit(_owner, _rewardManager, _lpToken, _rewardPoolId);
    }
```

Listing 3.4: BaseConnector.sol

Our analysis shows that the `initialize()` routine of the `BaseConnector` contract is not properly enforced with the above `initializer` modifier. Without it, this particular routine may be accidentally or intentionally invoked by others to disrupt the intended normal operations. In the same vein, the `initialize()` routine of the `syxOwnable` contract needs to be defined as `internal`, not `public`.

Recommendation Inherit the `Initializable` contract or the `VersionedInitializable` contract from OpenZeppelin for proper initialization with the required guarantee of executing the intended `initialize()` function only once during the entire lifetime. An example revision is show below:

```
    function initialize(
        address _owner,
        address _rewardManager,
        address _lpToken,
        uint8 _rewardPoolId
    ) external initializer {
        require(_owner != address(0), "ERR_OWNER_INVALID");
        require(_rewardManager != address(0), "ERR_REWARD_MANAGER");
        require(_lpToken != address(0), "ERR_LP_TOKEN");

        syxOwnable.initialize(_owner);
        rewardManager = IRewardManager(_rewardManager);
        lpToken = _lpToken;
        rewardPoolId = _rewardPoolId;

```

```
        emit LogInit(_owner, _rewardManager, _lpToken, _rewardPoolId);
    }
```

Listing 3.5: BaseConnector.sol

Status The issue has been fixed by this commit: `f49a29c28b25b048dbe4fada093fbd72d9b9ebf0`.

3.3 Duplicate Pool Detection and Prevention

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `RewardManager`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

The `Symblox` provides incentive mechanisms that reward the staking of `Balancer Pool` LP or `WVLX` tokens with `SYX` tokens. The rewards are carried out by designating a number of staking pools into which `Balancer Pool` LP or `WVLX` tokens can be staked. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards these stakers in a pool will receive are proportional to the amount of LP tokens they have staked in the pool versus the total amount of LP tokens staked in the pool.

In current implementation, there are two pools that share the rewarded `SYX` tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, `Symblox` has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock
```

```
            ? block.number
            : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accSyxPerShare: 0
            })
        );
    }
```

Listing 3.6: RewardManager.sol

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accSushiPerShare: 0
        }));
    }
```

Listing 3.7: RewardManager.sol (revised)

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status The issue has been fixed by this commit: `42961aed2fa28cdef876f2fedfa9fef09a28bf68`.

3.4 Recommended Explicit Pool Validity Checks

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `RewardManager`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

The `Symblox Yield Farming` has a central contract – `RewardManager` that has been tasked with the pool management, staking/unstaking support, as well as the reward distribution to various pools and stakers. In the following, we show the key `pool` data structure. Note all added pools are maintained in an array `poolInfo`.

```
    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool. SUSHIs to distribute per block.
        uint256 lastRewardBlock;  // Last block number that SUSHIs distribution occurs.
        uint256 accSushiPerShare; // Accumulated SUSHIs per share, times 1e12. See below.
    }
    ...
    // Info of each pool.
    PoolInfo[] public poolInfo;
```

Listing 3.8: RewardManager.sol

When there is a need to add a new pool, set a new `allocPoint` for an existing pool, stake (by depositing the supported `Balancer Pool` LP or `WVLX` tokens), unstake (by redeeming previously deposited `Balancer Pool` LP or `WVLX` tokens), query pending `SYX` rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range `[0, poolInfo.length-1]`. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validatePool`. This new modifier essentially ensures the given `_pool_id` or `_pid` indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
    /**
     * Deposit LP tokens to RewardManager for Symblox allocation.
     * @param _pid Reward pool Id
     * @param _amount Amount of LP tokens to deposit
     * @return Total amount of the user's LP tokens
     */
```

```
    function deposit(uint256 _pid, uint256 _amount) public returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user
                .amount
                .mul(pool.accSyxPerShare)
                .div(1e12)
                .sub(user.rewardDebt);
            safeSyxTransfer(msg.sender, pending);
        }
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSyxPerShare).div(1e12);
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        emit Deposit(msg.sender, _pid, _amount);
        return user.amount;
    }
```

Listing 3.9: RewardManager.sol

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including `set()`, `deposit()`, `withdraw()`, `emergencyWithdraw()`, `pendingSyx()` and `updatePool()`.

Recommendation Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
     modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "RewardManager: pool exists?");
        _;
    }

    // Deposit LP tokens to RewardManager for Symblox allocation.
    function deposit(uint256 _pid, uint256 _amount) public returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user
                .amount
                .mul(pool.accSyxPerShare)
                .div(1e12)
                .sub(user.rewardDebt);
            safeSyxTransfer(msg.sender, pending);
        }
```

```
190         user.amount = user.amount.add(_amount);
191         user.rewardDebt = user.amount.mul(pool.accSyxPerShare).div(1e12);
192         pool.lpToken.safeTransferFrom(
193             address(msg.sender),
194             address(this),
195             _amount
196         );
197         emit Deposit(msg.sender, _pid, _amount);
198         return user.amount;
199     }
```

Listing 3.10: RewardManager.sol

Status The issue has been fixed by this commit: `b60db878a9c7a4fce4b4183aa770cddf83122df4`.

3.5 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `RewardManager`
- Category: Business Logics [9]
- CWE subcategory: CWE-708 [5]

Description

In `Symblox`, the `RewardManager` contract operates as the main entry for interaction with staking users. The staking users `deposit()` the supported `Balancer Pool` LP or `WVLX` tokens into the pool and in return get proportionate share of the pool's rewards. Later on, the staking users can `withdraw()` their own assets from the pool. With assets in the pool, users can earn whatever incentive mechanisms proposed or adopted via governance.

Naturally, the above two functions, i.e., `deposit()` and `withdraw()`, are involved in transferring users' assets into or out of the `Symblox` protocol. Using the `deposit()` function as an example, it needs to transfer deposited assets from the user account to the pool (lines 192 – 196). When transferring standard ERC20 tokens, these asset-transferring routines work as expected: namely the account's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts (line 190).

```
172     /**
173      * Deposit LP tokens to RewardManager for Symblox allocation.
174      * @param _pid Reward pool Id
175      * @param _amount Amount of LP tokens to deposit
176      * @return Total amount of the user's LP tokens
177      */
178     function deposit(uint256 _pid, uint256 _amount) public returns (uint256) {
```

```
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user
                .amount
                .mul(pool.accSyxPerShare)
                .div(1e12)
                .sub(user.rewardDebt);
            safeSyxTransfer(msg.sender, pending);
        }
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSyxPerShare).div(1e12);
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        emit Deposit(msg.sender, _pid, _amount);
        return user.amount;
    }
```

Listing 3.11: RewardManager.sol

However, in the cases of deflationary tokens, as shown in the above code snippets, the input amount may not be equal to the received amount due to the charged transaction fee. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens.

One mitigation is to query the asset change right before and after the asset-transferring routines. In other words, instead of automatically assuming 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()`/`transferFrom()` is expected and aligned well with the intended operation. Though these additional checks cost additional gas usage, we feel that 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 `Symblox` pools. With the single entry of adding a new pool (via `add()`), `RewardManager` is indeed in the position to effectively regulate the set of assets allowed into the protocol.

Fortunately, both `Balancer Pool` LP and `WVLX` tokens are not deflationary/rebasing tokens and there is no need to take any action in `Symblox`. However, it is a potential risk if the current code base is used elsewhere or the need to add other tokens arises (e.g., in listing new DEX pairs).

Meanwhile, we need to point out that the forked `Balancer` shares the same issue in not supporting deflationary/rebasing tokens. In fact, there exist an earlier June attack [14] that has exploited this weakness for financial gains.

Recommendation Regulate the set of LP tokens supported in `Symblox` (including the forked `Balancer`) and, if there is a need to support deflationary tokens, add necessary mitigation mechanisms to keep track of accurate balances.

Status This issue has been confirmed. As there is a central place to regulate the assets that can be introduced into the protocol, the team decides no change for the time being.

3.6 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardManager`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [4]

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` [21] exploit, and the recent `Uniswap/Lendf.Me` hack [19].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `RewardManager` as an example, the `emergencyWithdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 234) starts before effecting the update on internal states (lines 236−237), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `emergencyWithdraw()` function.

```
235     // Withdraw without caring about rewards. EMERGENCY ONLY.
236     function emergencyWithdraw(uint256 _pid) public {
237         PoolInfo storage pool = poolInfo[_pid];
238         UserInfo storage user = userInfo[_pid][msg.sender];
239         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
240         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
```

```
        user.amount = 0;
        user.rewardDebt = 0;
    }
```

Listing 3.12: RewardManager.sol

Another similar violation can be found in the `deposit()` and `withdraw()` routines within the same contract.

```
    function deposit(uint256 _pid, uint256 _amount) public returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user
                .amount
                .mul(pool.accSyxPerShare)
                .div(1e12)
                .sub(user.rewardDebt);
            safeSyxTransfer(msg.sender, pending);
        }
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accSyxPerShare).div(1e12);
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        emit Deposit(msg.sender, _pid, _amount);
        return user.amount;
    }

    function withdraw(uint256 _pid, uint256 _amount) public returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accSyxPerShare).div(1e12).sub(
            user.rewardDebt
        );
        safeSyxTransfer(msg.sender, pending);
        user.amount = user.amount.sub(_amount);
        user.rewardDebt = user.amount.mul(pool.accSyxPerShare).div(1e12);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit Withdraw(msg.sender, _pid, _amount);
        return user.amount;
    }
```

Listing 3.13: RewardManager.sol

In the meantime, we should mention that the `Balancer`'s LP tokens implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions best practice. An example revision on the emergencyWithdraw routine is shown below:

```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 _amount=user.amount
        user.amount = 0;
        user.rewardDebt = 0;
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit EmergencyWithdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.14: RewardManager.sol (revised)

Status The issue has been fixed by this commit: 6facd88f1e78de42c7beec50919d47c5c07f8928.

3.7 Improved Logic in getMultiplier()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: RewardManager
- Category: Business Logics [9]
- CWE subcategory: CWE-708 [5]

Description

The RewardManager contract incentivizes early adopters by specifying an initial list of two supported pools into which early adopters can stake the supported Balancer's LP or WVLX tokens. The earnings were started at the specified startBlock. For every new block, there will be syxPerBlock new tokens minted and these minted tokens will be accordingly redistributed to the stakers of each pool. The rewarding will terminate at the endBlock height. For the initial blocks before bonusEndBlock, the amount of SYX tokens produced will be multiplied by 3 (specified in BONUS_MULTIPLIER), resulting in additional SYX tokens being minted and rewarded per block.

The early incentives are greatly facilitated by a helper function called getMultiplier(). This function takes two arguments, i.e., _from and _to, and calculates the reward multiplier for the given block range ([_from, _to]).

```
    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to)
        public
        view
        returns (uint256)
```

```
    {
        uint256 _endBlock;
        uint256 _startBlock;

        if (_to > endBlock) {
            _endBlock = endBlock;
        } else {
            _endBlock = _to;
        }

        if (_from > endBlock) {
            _startBlock = endBlock;
        } else {
            _startBlock = _from;
        }

        if (_endBlock <= bonusEndBlock) {
            return _endBlock.sub(_startBlock).mul(BONUS_MULTIPLIER);
        } else if (_startBlock >= bonusEndBlock) {
            return _endBlock.sub(_startBlock);
        } else {
            return
                bonusEndBlock.sub(_startBlock).mul(BONUS_MULTIPLIER).add(
                    _endBlock.sub(bonusEndBlock)
                );
        }
    }
```

Listing 3.15: RewardManager.sol

For elaboration, the helper's code snippet is shown above. We notice that this helper does not take into account the initial block (`startBlock`) from which the inventive rewards start to apply. As a result, when a normal user gives arbitrary arguments, it could return wrong reward multiplier! A correct implementation needs to take `startBlock` into account and appropriately re-adjusts the starting block number, i.e., `_from = _from >= startBlock ? _from : startBlock`.

We also notice that the helper function is called by two other routines, e.g., `pendingSyx()` and `updatePool()`. Fortunately, these two routines have ensured `_from >= startBlock` and always use the correct reward multiplier for reward redistribution.

Recommendation Apply additional sanity checks in the `getMultiplier()` routine so that the internal `_from` parameter can be adjusted to take `startBlock` into account.

```
    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) {
        uint256 _endBlock;
        uint256 _startBlock;

        _from = _from >= startBlock ? _from : startBlock;
        if (_to > endBlock) {
            _endBlock = endBlock;
```

```
        } else {
            _endBlock = _to;
        }

        if (_from > endBlock) {
            _startBlock = endBlock;
        } else {
            _startBlock = _from;
        }

        if (_endBlock <= bonusEndBlock) {
            return _endBlock.sub(_startBlock).mul(BONUS_MULTIPLIER);
        } else if (_startBlock >= bonusEndBlock) {
            return _endBlock.sub(_startBlock);
        } else {
            return
                bonusEndBlock.sub(_startBlock).mul(BONUS_MULTIPLIER).add(
                    _endBlock.sub(bonusEndBlock)
                );
        }
    }
```

Listing 3.16: RewardManager.sol

Status The issue has been fixed by this commit: `510feaa824457c721951fa31e5a4f324d11fa78c`.

3.8 Lack of Native Token Support in gulp()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BPool`
- Category: Business Logics [9]
- CWE subcategory: CWE-708 [5]

Description

As mentioned in Section 3.5, `Symblox` includes a forked version of `Balancer` with necessary customizations to support the trading of native tokens. The customizations include the additions of several routines, including `swapWTokenAmountIn()`, `swapExactAmountInWTokenOut()`, `joinswapWTokenIn()`, and `exitswapPoolAmountInWTokenOut()`, and the removals of `joinPool()`, `joinswapPoolAmountOut()`, and `exitswapExternAmountOut()`. Moreover, the `constructor()` routine of the `BPool` contract has been accordingly revised to initialize the wrapped token address, i.e., `wToken`. The naming of these functions is consistent with others in the default `Balancer` codebase.

While examining the customizations made in `Symblox`, we notice the `gulp()` routine has not been updated to support the wrapping of native tokens. As the native token has been wrapped in `Symblox` and the wrapped token gains the same level of trading or swapping as other tokens, we recommend the support of native tokens in `gulp()`.

```
    // Absorb any tokens that have been sent to this contract into the pool
    function gulp(address token) external _logs_ _lock_ {
        require(_records[token].bound, "ERR_NOT_BOUND");
        _records[token].balance = IERC20(token).balanceOf(address(this));
    }
```

Listing 3.17: BPool.sol

Recommendation Consider the need and accordingly add the native token support in the `gulp()` routine.

Status The issue has been fixed by this commit: `efd731316de859a305a4ae493e08f2f7697c9e90`.

4 Conclusion

In this audit, we have analyzed the implementation of the `Yield Farming` support in `Symblox`. `Symblox` itself is a cross-chain synthetic asset issuance and derivatives trading protocol that allows users to stake any supported tokens to mint synthetic assets. The audited system presents a unique addition to current DeFi offerings in maximizing yields for users. The current code base is clearly organized and those identified issues are promptly confirmed and fixed. As a final precaution, 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [13, 15, 16, 17, 20].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [22] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

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

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

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

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

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

[13] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[14] PeckShield. Balancer Hacks: Root Cause and Loss Analysis. https://blog.peckshield.com/2020/06/28/balancer.

[15] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[16] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[17] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

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

[20] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

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

[22] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.