# SMART CONTRACT AUDIT REPORT

for

# YSLv2

Prepared By: Xiaomi Huang

PeckShield

May 11, 2023

## Document Properties

| | |
|---|---|
| Client | YSL.IO |
| Title | Smart Contract Audit Report |
| Target | YSLv2 |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Patrick Liu, Jing Wang, Xuxian Jiang |
| Reviewed by | Patrick Liu |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | May 11, 2023 | Xuxian Jiang | Final Release |
| 1.0-rc1 | May 7, 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** 4
- 1.1 About YSLv2 . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

**2 Findings** 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

**3 Detailed Results** 11
- 3.1 Incorrect Share Amount Calculation in Various Vaults . . . 11
- 3.2 Revisited restrictTransfer Logic in Current Vaults . . . 13
- 3.3 Possible Withdrawal Prevention with lastTimestamp Update . . . 15
- 3.4 Sybil Attacks to Drain Vault Rewards . . . 16
- 3.5 Unauthorized Deposit in Multiple Vaults . . . 17
- 3.6 Incorrect USDy Price Calculation in Tokens/USDy . . . 19
- 3.7 Possible Sandwich Attacks to Manipulate Buyback Setting . . . 20
- 3.8 Incorrect Approve Amount in xYSLUSDCVault::_tax() . . . 22
- 3.9 Improper tokenHoldersCount Accounting in Receipt . . . 23
- 3.10 Missing Parameter Validation in YSL/xYSL . . . 24
- 3.11 Revisited Logic in PhoenixApeNFT::_securityCheck() . . . 26
- 3.12 Trust Issue of Admin Keys . . . 27

**4 Conclusion** 29

**References** 30

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the YSLv2 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 YSLv2

YSLv2 aims to optimize and amplify the returns from yield farming platforms through the maximization of locked liquidity. The protocol has a distinctive token economy and involves a series of tokens that all work in concert to create a dynamic ecosystem. It includes 1) YSL to serve as the protocols utility token; 2) xYSL to create locked liquidity with each transaction whilst also decreasing in supply, given its deflationary nature; 3) bYSL to serve as the governance token for the ecosystem; and 4) USDy to act as an reward token. Each token within the YSL.IO ecosystem presents a different value proposition and helps create a unique token economy. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The YSL.IO Protocol

| Item | Description |
|---:|---|
| Name | YSL.IO |
| Website | https://ysl.io/ |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | May 11, 2023 |

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

- https://github.com/ysl-io/ysl-protocol-v2.git (8a16af4)

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

- https://github.com/ysl-io/ysl-protocol-v2.git (6815808)

## 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.

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 |

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 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 YSLv2 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
| --- | --- | --- |
| Critical | 0 | |
| High | 3 | ■■■ |
| Medium | 4 | ■■■■ |
| Low | 5 | ■■■■■ |
| Informational | 0 | |
| Total | 12 | |

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 3 high-severity vulnerabilities, 4 medium-severity vulnerabilities, and 5 low-severity vulnerabilities.

Table 2.1: Key YSLv2 Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Incorrect Share Amount Calculation in Various Vaults | Business Logic | Resolved |
| PVE-002 | Medium | Revisited restrictTransfer Logic in Current Vaults | Business Logic | Resolved |
| PVE-003 | Medium | Possible Withdrawal Prevention with lastTimestamp Update | Business Logic | Resolved |
| PVE-004 | High | Sybil Attacks to Drain Vault Rewards | Business Logic | Resolved |
| PVE-005 | High | Unauthorized Deposit in Multiple Vaults | Business Logic | Resolved |
| PVE-006 | High | Incorrect USDy Price Calculation in Tokens/USDy | Business Logic | Resolved |
| PVE-007 | Low | Possible Sandwich Attacks to Manipulate Buyback Setting | Time And State | Resolved |
| PVE-008 | Low | Incorrect Approve Amount in xYSLUSDCVault::_tax() | Business Logic | Resolved |
| PVE-009 | Low | Improper tokenHoldersCount Accounting in Receipt | Business Logic | Resolved |
| PVE-010 | Low | Missing Parameter Validation in YSL/xYSL | Coding Practices | Resolved |
| PVE-011 | Medium | Revisited Logic in PhoenixApeNFT::_securityCheck() | Business Logic | Resolved |
| PVE-012 | Medium | Trust Issue Of Admin Keys | Security Features | Mitigated |

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

# 3 | Detailed Results

## 3.1 Incorrect Share Amount Calculation in Various Vaults

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The `YSLv2` protocol has a number of vaults. And users can deposit their tokens into the chosen vault and get the vault share/receipt in return. While reviewing the vault share calculation, we notice the current approach needs to be revisited.

For elaboration, we show below an example vault – `YSLVault` – and its `deposit()` routine. As the name indicates, this function allows for the staking to obtain the respective vault share. However, when `depositTax=0` (lines 176-177), the new vault share is computed as `tokenAmount = _amount`, which is incorrect. The correct share amount should be calculated as follows: `amount * IReceipt(Admin.YSL()).totalSupply())/ (totalDeposit- amount)`. Note this issue affects all existing vaults.

```
134     function deposit(
135         address _user,
136         uint256 _amount
137     )
138         external
139         nonReentrant
140         whenNotPaused
141         _securityCheck(_user)
142         _checkPerpetualRatioIncreased
143     {
144         require(
145             _user != address(0),
146             "YSL Vault: The user address cannot be set to 0x0."
147         );
148         require(
```

```
            _amount > 0,
            "YSL Vault: The amount must be greater than zero."
        );
        if (IReceipt(Admin.YSLS()).totalSupply() != 0) {
            if (pendingRewards(_user) > 0) {
                _claimReward(_user);
            } else {
                currentRewardPerShare[_user] = rewardPerShare;
            }
        }
        IERC20(Admin.YSL()).transferFrom(_user, address(this), _amount);
        if (IReceipt(Admin.YSLS()).totalSupply() == 0) {
            require(
                msg.sender == Admin.teamAddress(),
                "YSL Vault: Only the team can deposit first."
            );
            IReceipt(Admin.YSLS()).mint(_user, _amount);
            perpetualRatio = 10**18;
        } else {
            uint256 tokenAmount;
            if (depositTax > 0) {
                uint256 taxedAmount = (_amount * (100 - depositTax)) / 100;
                uint256 ratio = (taxedAmount * 1e18) /
                    IERC20(Admin.YSL()).balanceOf(address(this));
                tokenAmount =
                    (ratio * IReceipt(Admin.YSLS()).totalSupply()) /
                    (1e18 - ratio);
            } else {
                tokenAmount = _amount;
            }
            IReceipt(Admin.YSLS()).mint(_user, tokenAmount);
            perpetualRatio =
                (IERC20(Admin.YSL()).balanceOf(address(this)) * (10**18)) /
                IReceipt(Admin.YSLS()).totalSupply();
        }
        restrictTransfer[msg.sender] = block.number;
        emit Deposit(
            "YSL Vault",
            address(this),
            msg.sender,
            _amount,
            block.number,
            block.timestamp
        );
    }
```

Listing 3.1: `YSLVault::deposit()`

**Recommendation** Revisit the above deposit logic to compute and mint the right vault share.

**Status** The issue has been resolved in the following commit: `dfcd809`.

## 3.2 Revisited restrictTransfer Logic in Current Vaults

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

For each vault supported in `YSLv2`, there is a setting to restrict unwanted frequent transfers. This restriction is enforced by keeping track of an user's last active block number in `restrictTransfer`. While examining this setting, we notice the current logic needs to be improved.

Specifically, we show below the setting enforcement in `xYSLVault`. For each deposit operation, the user may be restricted in performing only once per block. This restriction is enforced with the `_securityCheck(_user)` modifier (line 141) as the user's last active block number is updated in `restrictTransfer[msg.sender] = block.number` (line 182). It comes to our attention that the restriction should be enforced for the given `user`, not `msg.sender`. Note this issue affects a number of existing vaults.

```
134     function deposit(
135         address _user,
136         uint256 _amount
137     )
138         external
139         nonReentrant
140         whenNotPaused
141         _securityCheck(_user)
142         _checkPerpetualRatioIncreased
143     {
144         require(
145             _user != address(0),
146             "xYSL Vault: The user address cannot be set to 0x0."
147         );
148         require(
149             _amount > 0,
150             "xYSL Vault: The amount must be greater than zero."
151         );
152         if (pendingRewards(_user) > 0) {
153             _claimReward(_user);
154         } else {
155             currentRewardPerShare[_user] = rewardPerShare;
156         }
157         IERC20(Admin.xYSL()).transferFrom(_user, address(this), _amount);
158         if (IReceipt(Admin.xYSLS()).totalSupply() == 0) {
159             require(
```

```
                msg.sender == Admin.teamAddress(),
                "xYSL Vault: Only the team can deposit first."
            );
            IReceipt(Admin.xYSLS()).mint(_user, _amount);
            perpetualRatio = 10**18;
        } else {
            uint256 tokenAmount;
            if (depositTax > 0) {
                uint256 taxedAmount = (_amount * (100 - depositTax)) / 100;
                uint256 ratio = (taxedAmount * 1e18) /
                    IERC20(Admin.xYSL()).balanceOf(address(this));
                tokenAmount =
                    (ratio * IReceipt(Admin.xYSLS()).totalSupply()) /
                    (1e18 - ratio);
            } else {
                tokenAmount = _amount;
            }
            IReceipt(Admin.xYSLS()).mint(_user, tokenAmount);
            perpetualRatio =
                (IERC20(Admin.xYSL()).balanceOf(address(this)) * (10**18)) /
                IReceipt(Admin.xYSLS()).totalSupply();
        }
        restrictTransfer[msg.sender] = block.number;
        emit Deposit(
            "xYSL Vault",
            address(this),
            msg.sender,
            _amount,
            block.number,
            block.timestamp
        );
    }
```

Listing 3.2: `xYSLVault::deposit()`

**Recommendation** Revise the above logic to properly enforce unwanted frequent transfers.

**Status** The issue has been resolved in the following commit: `292a8c4`.

## 3.3 Possible Withdrawal Prevention with lastTimestamp Update

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The `YSLv2` protocol has developed a number of anti-dump mechanisms. One specific one is to restrict the withdrawal so that the staked tokens may not be withdrawn until 4 epochs have passed. Our analysis shows this restriction may be abused to mount a denial-of-service against a staking user.

To elaborate, we show below an example withdrawal logic in `YSLVault`, which enforces the following requirement: `require(block.timestamp - IReceipt(Admin.YSLS()).lastTimestamp(_user)>=4 * Admin.epochDuration())` (lines 229-233). However, it comes to our attention that an user's `lastTimestamp` state may be updated to current block timestamp by simply sending a dust amount to the victim user (line 248)!

```
205     function withdraw(
206         address _user,
207         uint256 _amount,
208         address _sendTo
209     )
210         external
211         nonReentrant
212         whenNotPaused
213         _securityCheck(_user)
214         _checkPerpetualRatioIncreased
215     {
216         uint256 userYSLSBalance = IReceipt(Admin.YSLS()).balanceOf(_user);
217         require(
218             _user != address(0),
219             "YSL Vault: The user address cannot be set to 0x0."
220         );
221         require(
222             _amount > 0,
223             "YSL Vault: The amount must be greater than zero."
224         );
225         require(
226             userYSLSBalance >= _amount,
227             "YSL Vault: Insufficient receipt tokens for withdrawal."
228         );
229         require(
230             block.timestamp - IReceipt(Admin.YSLS()).lastTimestamp(_user) >=
231                 4 * Admin.epochDuration(),
```

```
            "YSL Vault: Withdrawal not allowed until 4 epochs have passed since last
                deposit or withdrawal."
        );
        ...
    }
```

Listing 3.3: `YSLVault::withdraw()`

```
    function _transfer(
        address _sender,
        address _recipient,
        uint256 _amount
    )
        internal
        override
        whenNotPaused
        securityCheck(_sender, _recipient)
    {
        if(balanceOf(_recipient) == 0 && _amount > 0){
            tokenHoldersCount++;
        }
        super._transfer(_sender, _recipient, _amount);
        if(balanceOf(_sender) == 0){
            tokenHoldersCount--;
        }
        lastTimestamp[_recipient] = block.timestamp;
    }
```

Listing 3.4: `Receipt::_transfer()`

**Recommendation** Revise the above withdrawal restriction logic to eliminate the possible denial-of-service risk.

**Status** The issue has been resolved in the following commit: `567a0f1`.

## 3.4 Sybil Attacks to Drain Vault Rewards

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Receipt`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

### Description

In `YSLv2`, the `Receipt` contract maintains the vault share, which is used to compute possible rewards. The `Receipt` contract is implemented as an ERC20 token, which can be transferred to others. However, the transfer logic does not properly claim the rewards fro the sender. And this issue may be exploited to drain vault rewards.

In the following, we use the YSLVault as an example. The calculation of pending rewards for a given user is based on two factors: the user's balance IReceipt(Admin.YSLS()).balanceOf(_user) and the increase of rewardPerShareForUser. The latter may be manipulated by launching a so-called Sybil attack as the new user basically has 0 in its currentRewardPerShare[_user].

```
    function pendingRewards(address _user)
        public
        view
        returns (uint256 _reward)
    {
        if (IReceipt(Admin.YSLS()).totalSupply() > 0) {
            uint256 rewardPerShareForUser = rewardPerShare -
                currentRewardPerShare[_user];
            _reward =
                (IReceipt(Admin.YSLS()).balanceOf(_user) *
                    rewardPerShareForUser) /
                1e18;
        }
    }
```

Listing 3.5: YSLVault::pendingRewards()

**Recommendation** To mitigate, it is necessary to accompany every single transfer() and transferFrom() in Receipt to proactively keep track of the currentRewardPerShare for each involved user.

**Status** The issue has been resolved in the following commit: 567a0f1.

## 3.5 Unauthorized Deposit in Multiple Vaults

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: Receipt
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

### Description

As mentioned earlier, for each vault, users can deposit their tokens and get the vault share/receipt in return. While reviewing the current deposit logic, we notice the funds are transferred from the given user, not the calling user.

In the following, we use the `YSLVault` as an example. The funding source for the deposit is the input `_user` argument, which may not have authorized the calling user to make the transfer! As a result, a malicious user may force other victim users to deposit the funds into the vault, even the victim may not plan to do so!

```
    function deposit(
        address _user,
        uint256 _amount
    )
        external
        nonReentrant
        whenNotPaused
        _securityCheck(_user)
        _checkPerpetualRatioIncreased
    {
        require(
            _user != address(0),
            "YSL Vault: The user address cannot be set to 0x0."
        );
        require(
            _amount > 0,
            "YSL Vault: The amount must be greater than zero."
        );
        if (IReceipt(Admin.YSLS()).totalSupply() != 0) {
            if (pendingRewards(_user) > 0) {
                _claimReward(_user);
            } else {
                currentRewardPerShare[_user] = rewardPerShare;
            }
        }
        IERC20(Admin.YSL()).transferFrom(_user, address(this), _amount);
        if (IReceipt(Admin.YSLS()).totalSupply() == 0) {
            require(
                msg.sender == Admin.teamAddress(),
                "YSL Vault: Only the team can deposit first."
            );
            IReceipt(Admin.YSLS()).mint(_user, _amount);
            perpetualRatio = 10**18;
        } else {
            uint256 tokenAmount;
            if (depositTax > 0) {
                uint256 taxedAmount = (_amount * (100 - depositTax)) / 100;
                uint256 ratio = (taxedAmount * 1e18) /
                    IERC20(Admin.YSL()).balanceOf(address(this));
                tokenAmount =
                    (ratio * IReceipt(Admin.YSLS()).totalSupply()) /
                    (1e18 - ratio);
            } else {
                tokenAmount = _amount;
            }
            IReceipt(Admin.YSLS()).mint(_user, tokenAmount);
            perpetualRatio =
```

```
                (IERC20(Admin.YSL()).balanceOf(address(this)) * (10**18)) /
                IReceipt(Admin.YSLS()).totalSupply();
        }
        restrictTransfer[msg.sender] = block.number;
        emit Deposit(
            "YSL Vault",
            address(this),
            msg.sender,
            _amount,
            block.number,
            block.timestamp
        );
    }
```

Listing 3.6: `YSLVault::deposit()`

**Recommendation** Revise the above logic to ensure the funds are provided by the calling user `msg.sender`.

**Status** To address this issue, the team has implemented a security check modifier that verifies whether the `msg.sender` is whitelisted or the same as the user. This logic is necessary to allow users to call the `claimStakeAll()` function and deposit funds on behalf of another user.

## 3.6 Incorrect USDy Price Calculation in Tokens/USDy

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: `USDy`
- Category: Business Logics [5]
- CWE subcategory: CWE-841 [3]

### Description

In `YSLv2`, the `USDy` token is an reward token which comes with an inherent mint price. While examining the current approach to compute the mint price, we notice the logic is flawed.

In the following, we show below its implementation in `mintPrice()`. This routine has a rather straightforward logic in returning the current price of the token in terms of USDC. However, the computation is based on the instant trading pair price from the `USDC-USDy` trading pair. However, the given swap path for price calculation should be `USDy -> USDC`, instead of current `USDC -> USDy` (lines 287-288).

```
281    function mintPrice()
282        external
283        view
```

```
        returns (uint256)
    {
        address[] memory path = new address[](2);
        path[0] = Admin.USDC();
        path[1] = address(this);
        uint256 poolPriceUSDy = IApeRouter02(Admin.apeswapRouter())
            .getAmountsOut(1 * 10**18, path)[1];
        if (protocolPriceUSDy < poolPriceUSDy) {
            return poolPriceUSDy;
        } else {
            return protocolPriceUSDy;
        }
    }
```

Listing 3.7: `USDy::mintPrice()`

**Recommendation** Revise the above routine to compute the correct mint price of `USDy`.

**Status** The issue has been resolved in the following commit: `fb34d31`.

## 3.7 Possible Sandwich Attacks to Manipulate Buyback Setting

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `YSLv2` protocol is no exception. Specifically, if we examine the `Admin` contract, it has defined a number of protocol-wide risk parameters, such as `buybackActivation` as well as a variety of vaults. In the following, we show the corresponding routine to update `buybackActivation`.

```
    function updateBuyback()
        external
        onlyAdminOrOperatorOrSetter
    {
        address[] memory path = new address[](2);
        path[0] = USDy;
        path[1] = USDC;
        if (buybackActivation) {
            if (
                buybackActivationEpoch + (4 * epochDuration) < block.timestamp
            ) {
                95 * 10**16 <=
```

```
                    IApeRouter02(apeswapRouter).getAmountsOut(10**18, path)[1]
                    ? setBuybackActivation(false)
                    : setBuybackActivationEpoch();
            }
        } else if (
            buybackActivationEpoch + (4 * epochDuration) < block.timestamp &&
            95 * 10**16 >
            IApeRouter02(apeswapRouter).getAmountsOut(10**18, path)[1]
        ) {
            setBuybackActivation(true);
            setBuybackActivationEpoch();
        }
    }
```

Listing 3.8: `Admin::updateBuyback()`

From the above routine, we notice the `buybackActivation` setting may be based on the current instant trading pair price (via `apeswapRouter.getAmountsOut()`. However, this function simply relies on the pair's reserves for the price calculation. Apparently, this approach to query for current price is highly unreliable and suffers from price manipulation!

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense. The same issue is also applicable to other routines behind extra `USDy` mints, which may be mitigatd by existing anti-dump mechanism.

**Recommendation** Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

**Status** This issue is confirmed by the team as part of the protocol's design. However, the team clarifies that various protocol restrictions prevent any harm to the protocol.

## 3.8 Incorrect Approve Amount in xYSLUSDCVault::_tax()

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `xYSLUSDCVault`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The `YSLv2` protocol has its own unique tokenomics. For example, the transfer of protocol tokens may come with the transfer tax. While examining the current tax collection in the `xYSLUSDCVault`, we notice the tax collection needs to be improved.

To elaborate, we show below the related `xYSLUSDCVault::_tax()` routine. When the `buybackActivation` setting is turned on, `0.1%` of collected tax will be sent to buy and burn `USDy`. However, the current tax collects approves `0.2%` of collected tax to `IApeRouter02(router)`! This approved amount needs to be aligned with the intended design tokenomics.

```
    function _tax(uint256 _amount) internal {
        address[] memory path = new address[](2);
        address USDC = Admin.USDC();
        address USDy = Admin.USDy();
        address router = Admin.apeswapRouter();
        path[0] = USDy;
        path[1] = USDC;
        IERC20(USDC).safeTransfer(Admin.treasury(), (_amount * 75) / 1000);
        if (Admin.buybackActivation()) {
            Admin.updateBuyback();
            IERC20(USDC).safeTransfer(
                Admin.teamAddress(),
                (_amount * 10 * 25) / 10000
            );
            path[0] = USDC;
            path[1] = USDy;
            IERC20(USDC).safeApprove(router, (_amount * 20 * 75) / 10000);
            uint256 amountOut = IApeRouter02(router).swapExactTokensForTokens(
                (_amount * 10 * 75) / 10000,
                0,
                path,
                address(this),
                block.timestamp + 1000
            )[path.length - 1];
            IReceipt(USDy).burn(address(this), amountOut);
        } else {
            Admin.updateBuyback();
            IERC20(USDC).safeApprove(Admin.referral(), _amount / 10);
            IReferral(Admin.referral()).rewardDistribution(
```

```
                address(this),
                msg.sender,
                _amount / 10
            );
        }
    }
```

Listing 3.9: `xYSLUSDCVault::_tax()`

**Recommendation** Revisit the above logic to approve only the intended amount for swap. The same issue is also applicable to another routine `xYSLUSDCVault::_deposit()`.

**Status** The issue has been resolved in the following commit: `dfcd809`.

## 3.9 Improper tokenHoldersCount Accounting in Receipt

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Receipt`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The staking share is represented as the ERC20-compliant `Receipt` token. This `Receipt` token is also enhanced with the number of total holders in `tokenHoldersCount`. Our analysis shows this `tokenHoldersCount` state can be accurately recorded.

To elaborate, we show below the related `_transfer` routine. This routine will be invoked for each transfer even when the amount is equal to 0. However, the current logic will decrement the `tokenHoldersCount` number by 1 if the sender has 0 balance and the transfer amount is 0. To improve, we need to adjust the accounting by ensuring that the number will not be decremented by 1 unless the the sender balance becomes 0 after transfering non-0 amount.

```
    function _transfer(
        address _sender,
        address _recipient,
        uint256 _amount
    )
        internal
        override
        whenNotPaused
        securityCheck(_sender, _recipient)
    {
        if(balanceOf(_recipient) == 0 && _amount > 0){
            tokenHoldersCount++;
```

```
        }
        super._transfer(_sender, _recipient, _amount);
        if(balanceOf(_sender) == 0){
            tokenHoldersCount--;
        }
        lastTimestamp[_recipient] = block.timestamp;
    }
```

Listing 3.10: Receipt::_transfer()

**Recommendation** Revise the above logic to properly keep track of the `tokenHoldersCount` state.

**Status** The team has added a check to the `burnBlacklistToken()` function to prevent an unnecessary decrease in the token holder's count. This is because the function can be called for a user with a zero balance, and decrementing the count in such cases would not accurately reflect the actual number of token holders.

## 3.10 Missing Parameter Validation in YSL/xYSL

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

As mentioned in Section 3.7, DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `YSLv2` protocol is no exception. Specifically, if we examine the `bYSL` contract, it has defined a number of protocol-wide risk parameters, such as `USDyBuyback` and `bYSLTaxAllocation`. In the following, we show the corresponding routines that allow for their changes.

```
    function setUSDyBuyback(uint256[] memory _USDyBuyback)
        external
        onlyAdmin
    {
        emit SetUSDyBuyback(
            "bYSL",
            address(this),
            USDyBuyback,
            _USDyBuyback,
            block.number,
            block.timestamp
        );
```

```
        USDyBuyback = _USDyBuyback;
    }

    /**
     * @notice Sets the address of the liquidity pool for the contract.
     * @param _lp: The address of the liquidity pool.
     * @dev This function can only be called by the contract owner and requires that the
         specified address is not the null address (0x0).
     * It also checks that the specified address is not already the current liquidity
        pool address.
     */
    function setLiquidityPool(address _lp)
        external
        onlyAdmin
    {
        require(
            _lp != address(0),
            "bYSL: The liquidity pool address cannot be set to 0x0."
        );
        emit SetterForAddress(
            "bYSL",
            address(this),
            "setLiquidityPool",
            liquidityPool,
            _lp,
            block.number,
            block.timestamp
        );
        liquidityPool = _lp;
    }
```

Listing 3.11: `bYSL::setUSDyBuyback()/setLiquidityPool()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `USDyBuyback` may allocate unreasonably high portion in the buyback payment, hence incurring cost to users or hurting the adoption of the protocol.

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

**Status** The issue has been resolved in the following commits: `bb0b066` and `88bafe6f`.

## 3.11 Revisited Logic in PhoenixApeNFT::_securityCheck()

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PhoenixApeNFT`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

### Description

The `YSLv2` protocol has a built-in `PhoenixApeNFT` contract that may be used for access control purposes. In particular, this NFT contract grants the holders certain privilege in accessing protocol-wide features. While examining this NFT contract, we notice one core `_securityCheck` modifier needs to be revised.

To elaborate, we show below its implementation. It has four different requirements. Among these four, the second one needs to be revisited. Specifically, the second requirement checks whether the given `_user` is in the blacklist. If yes, it requires the caller needs to have the `DEFAULT_ADMIN_ROLE` . With that, this requirement should be enforced as the following: `require(((permissionLists.checkBlacklistAddress(_user)&& hasRole(DEFAULT_ADMIN_ROLE, msg.sender))|| !permissionLists.checkBlacklistAddres (_user))` (lines 67-72).

```
63    modifier _securityCheck(address _user) {
64        IPermissionLists permissionLists = IPermissionLists(
65            adminContract.permissionLists()
66        );
67        require(
68            ((permissionLists.checkBlacklistAddress(_user) &&
69                hasRole(DEFAULT_ADMIN_ROLE, msg.sender))
70                !permissionLists.checkBlacklistAddress(msg.sender)),
71            "Phoenix Ape NFT: Sender address is blacklisted and cannot interact with
                this contract."
72        );
73        if (_isContract(msg.sender)) {
74            require(
75                permissionLists.checkWhitelistAddress(msg.sender),
76                "Phoenix Ape NFT: External contract address is not whitelisted and
                    cannot interact with this contract."
77            );
78        }
79        if (_user != msg.sender) {
80            if (_isContract(_user)) {
81                require(
82                    permissionLists.checkWhitelistAddress(_user),
83                    "Phoenix Ape NFT: External contract address is not whitelisted and
                        cannot interact with this contract."
84                );
```

```
            }
        }
        if (!permissionLists.checkWhitelistAddress(_user)) {
            require(
                restrictTransfer[_user] != block.number,
                "Phoenix Ape NFT: The provided user address is not whitelisted and
                    cannot interact with this contract within the same block."
            );
        }
        _;
    }
```

Listing 3.12: PhoenixApeNFT::_securityCheck()

**Recommendation** Revise the above `_securityCheck()` to achieve its intended purpose.

**Status** The issue has been resolved in the following commit: `6c0bd54`.

## 3.12 Trust Issue of Admin Keys

- ID: PVE-012
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

### Description

In the `YSLv2` protocol, there is a privileged `admin` account (with the `DEFAULT_ADMIN_ROLE` role) that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and fee adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setPermissionLists(address _permissionLists)
        external
        onlyAdmin
    {
        permissionLists = _permissionLists;
    }

    /**
     * @notice Allows the admin to set the PhoenixApeNFT address.
     * @param _phoenixApeNFT: The address of the PhoenixApeNFT contract.
     * @dev Function to set the PhoenixApeNFT address.
     */
    function setPhoenixApeNFT(address _phoenixApeNFT)
```

```
        external
        onlyAdmin
    {
        phoenixApeNFT = _phoenixApeNFT;
    }

    /**
     * @notice Allows the admin to set the PhoenixApeRental address.
     * @param _phoenixApeRental: The address of the PhoenixApeRental contract.
     * @dev Function to set the PhoenixApeRental address.
     */
    function setPhoenixApeRental(address _phoenixApeRental)
        external
        onlyAdmin
    {
        phoenixApeRental = _phoenixApeRental;
    }

    /**
     * @notice Allows the admin to set the YSL Swaps address.
     * @param _YSLSwaps: The address of the YSL Swaps contract.
     * @dev Function to set the YSL Swaps address.
     */
    function setYSLSwaps(address _YSLSwaps)
        external
        onlyAdmin
    {
        YSLSwaps = _YSLSwaps;
    }
```

Listing 3.13: Example Privileged Opeations in `Admin`

If the privileged `admin` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

**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** The team has confirmed that the admin keys will be protected by a `Gnosis Multisig`.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the YSLv2 protocol, which aims to optimize and amplify the returns from yield farming platforms through the maximization of locked liquidity. The protocol has a distinctive token economy and involves a series of tokens that all work in concert to create a dynamic ecosystem. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

# References


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

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.