SMART CONTRACT AUDIT REPORT

for

YSL.IO Protocol

Prepared By: Yiqun Chen

PeckShield
November 17, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	November 17, 2021	Xuxian Jiang	Final Release
1.0-rc1	November 7, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the YSL.IO 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 YSL.IO

YSL.IO aims to optimize and amplify the returns from yield farming platforms through the maximization of locked liquidity. The protocol has a distinctive token economy of three native tokens: YSL, sYSL, and xYSL. The first one serves as the protocols utility token; The second one is a dual-purpose token acting both as the reward token and the governance token for the YSL.IO ecosystem (and its price is determined not only by market activity but is also linked to the amount of YSL-BUSD locked liquidity); And the third token is designed to create locked liquidity with each transaction whilst also decreasing in supply, given its deflationary nature. Each token within the YSL.IO ecosystem presents a different value proposition and helps create a unique token economy.

The basic information of the YSL.IO 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	November 17, 2021

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.git (a0c856d)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

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

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

Table 1.3: The Full List of Check Items

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], 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 YSL.IO 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	2	■■
Medium	2	■■
Low	3	■■■
Informational	0	
Total	7	

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 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key YSL.IO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited Design on YSLToken::burnFrom()	Business Logic	Resolved
PVE-002	Low	Duplicate Functions in aYSLToken	Coding Practices	Resolved
PVE-003	High	Proper Airdrop Unlocking in sYSLToken	Business Logic	Resolved
PVE-004	Low	Early Return in xYSLToken::_afterTokenTransfer()	Business Logic	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-006	Medium	Unfair Pool Reward Update in YSLProtocol	Business Logic	Resolved
PVE-007	High	Possible Sandwich Attacks in sYSLShare	Time and State	Resolved

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

3 Detailed Results

3.1 Revisited Design on YSLToken::burnFrom()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: YSLToken
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The YSL.IO protocol has a distinctive token economy of three native tokens: YSL, sYSL, and xYSL. While reviewing the first token, we notice the presence of certain privileged accounts (e.g., minters), which may be able to mint (or burn) additional tokens into (or from) circulation. However, our analysis shows that the burn logic needs to be revisited.

For elaboration, we show below the related burnFrom() function. As the name indicates, this function allows for burning the tokens from the given account. Note this is a privileged operation and can be only called by the so-called minters. However, it comes to our attention that the current logic still requires the user to grant the allowance to the minter before the specified amount of tokens can be burnt.

```
    ///@notice Provides burning functionality
    ///@dev Available only for minter contract
    ///@param account Choosen account for burning
    ///@param amount Amount of tokens to burn
    function burnFrom(address account, uint256 amount) external _isMinter {
        transferFrom(account, _msgSender(), amount);
        _burn(_msgSender(), amount);
    }
```

Listing 3.1: YSLToken::burnFrom()

Recommendation Revisit the above burn logic to remove the dependency on the user approval.

Status The issue has been resolved as the team clarifies that the design is intended to require user approval to avoid centralization. In other words, the admin around `burnFrom()` is not privileged.

3.2 Duplicate Functions in aYSLToken

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `aYSLToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `YSL.IO` protocol has built-in three native tokens: `YSL`, `sYSL`, and `xYSL`. The current repository also has the fourth standalone token contract `aYSLToken`. While this standalone token contract has a rather standard ERC20 token implementation, we notice its implementation needs to be revised.

Specifically, the current implementation has two `constructor()` functions (as shown below) and two `_isMinter()` modifiers. Apparently, one of them needs to be properly removed.

```
    ///@notice Perform contaract initial setup
    constructor() ERC20("aYSL token", "aYSL") {
        _setupRole(MINTER_ROLE, _msgSender());
        _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
    }
```

Listing 3.2: `aYSLToken::constructor()`

Recommendation Remove the duplicate definitions in the `aYSLToken` contract. If this contract is not currently in use, we suggest to remove it from the current repository.

Status This issue has been fixed by removing the duplicate functions.

3.3 Proper Airdrop Unlocking in sYSLToken

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `sYSLToken`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The second native token `sYSLToken` in the protocol acts as the reward token and the governance token for the `YSL.IO` ecosystem and its price is determined not only by market activity but is also linked to the amount of `YSL-BUSD` locked liquidity. It has a public function `unlockAirdrop()` that is used to unlock airdropped tokens. However, its current implementation is flawed.

To elaborate, we show below its implementation. The airdropped tokens are linearly locked and follows the normal vesting schedule. However, the calculation of new unlocked amount mistakenly uses the `usersLocked[account].unlocked` (line 253), instead of the intended `usersAirdropLocked[account].unlocked`. As a result, it could be possible that more tokens may be mistakenly vested.

```
245     ///@notice Unlocks airdrop vested tokens (linearly locked)
246     ///@param account Address to unlock tokens for
247     function unlockAirdrop(address account) public {
248         LockedFunds memory lf = usersAirdropLocked[account];
249         if (lf.lockAmount == 0 || lf.unlocked == lf.lockAmount) {
250             return;
251         }
252
253         uint256 unlockedBefore = usersLocked[account].unlocked;
254         if (block.timestamp >= lf.lockBlockTimestamp + lf.lockTime) {
255             usersAirdropLocked[account].unlocked = lf.lockAmount;
256         } else {
257             usersAirdropLocked[account].unlocked =
258                 (lf.lockAmount * (block.timestamp - lf.lockBlockTimestamp)) /
259                 lf.lockTime;
260         }
261
262         if (transferredLockedAmount[account] > 0) {
263             uint256 unlockedAfter = usersAirdropLocked[account].unlocked;
264             uint256 diff = unlockedAfter - unlockedBefore;
265             if (transferredLockedAmount[account] >= diff) {
266                 transferredLockedAmount[account] -= diff;
267             } else {
268                 transferredLockedAmount[account] = 0;
269             }
270         }
271     }
```

Listing 3.3: `sYSLToken::unlockAirdrop()`

Recommendation Revise the above `unlockAirdrop()` logic by properly making use of the right `usersAirdropLocked` amount.

Status The issue has been confirmed. The team has performed a workaround to greatly contain the impact of this issue.

3.4 Early Return in xYSLToken::_afterTokenTransfer()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: xYSLToken
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The third native token xYSLToken is deflationary with the purpose of creating locked liquidity with each transaction (with ever-decreased supply). The deflationary nature is implemented in a helper routine _afterTokenTransfer(). However, the current implementation makes an early return without making the token deflationary.

To elaborate, we show below it full implementation. When any involved party, either from or to, is whitelisted, a transaction fee is imposed. However, the early return statement (line 409) avoids the charge of any transaction fee.

```
402     function _afterTokenTransfer(
403         address from,
404         address to,
405         uint256 amount
406     ) internal {
407         if (whitelist[from]  whitelist[to]) {
408             emit Transfer(from, to, amount);
409             return;
410             if (adapter == address(0)) {
411                 emit Transfer(from, to, amount);
412                 return;
413             }
414             uint256 fee = (amount * FEE) / FEE_DIVIDER;
415             uint256 feeBurned = (amount * FEE_BURN) / FEE_DIVIDER;
416             uint256 feeAdapter = fee - feeBurned;

418             _balances[to] -= fee;
419             _balances[address(0)] += feeBurned;

421             _balances[adapter] += feeAdapter;
422             IxYSLAdapter(adapter).transferSurcharge();

424             emit Transfer(from, to, amount - fee);
425             emit Transfer(from, address(0), feeBurned);
426             emit Transfer(from, adapter, feeAdapter);
427         }
428     }
```

Listing 3.4: xYSLToken::_afterTokenTransfer()

Recommendation Improve the above `_afterTokenTransfer()` logic to make the third token deflationary.

Status The issue has been fixed by avoiding the early return in `_afterTokenTransfer()`.

3.5 Trust Issue of Admin Keys

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

Description

In the `YSL.IO` 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.

```
    ///@notice Sets minter role
    ///@param _minter Address that will be set as minter
    function setMinter(address _minter) external onlyOwner {
        require(_minter != address(0), "Null address provided");
        _setupRole(MINTER_ROLE, _minter);
    }

    ///@notice Remove minter role
    ///@param _minter Address that will be removed as minter
    function removeMinter(address _minter) external onlyOwner {
        require(_minter != address(0), "Null address provided");
        revokeRole(MINTER_ROLE, _minter);
    }
```

Listing 3.5: Example Privileged Opeations in `YSLToken`

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 ownership will be transferred to a `Timelock` contract upon the deployment.

3.6 Unfair Pool Reward Update in YSLProtocol

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `YSLProtocol`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `YSL.IO` protocol has a built-in incentive mechanism which follows the `MasterChef` approach to disseminate the second token `sYSL`. By design, the protocol also reserves certain emission rewards for the team to support their development efforts. The related team-rewarding emission rate is supposed to be allocated from all active pools. However, our analysis shows that it directly consumes the pool who happens to be updated after one day of team-related emission.

To elaborate, we show below the related `udpatePool()` function. Notice the condition of `if (block.timestamp - lastTeamUpdate >= 1 days && teamRate > 0)` (line 204). If it is met, there is a need to mint the tokens to the team. However, the `teamRate` is directly reduced from the computed pool's `sYSLReward`. In other words, other pools are not affected for this round of team rewards. This may not be fair to the affected pool as the team-related rewards are supposed to be shared by all current pools.

```
188     /// @notice Update reward variables of the given asset to be up-to-date.
189     /// @param _pid Pool's id
190     function updatePool(uint256 _pid) public {
191         PoolInfo storage pool = poolInfo[_pid];
192         if (block.number <= pool.lastRewardBlock) {
193             return;
194         }
195         uint256 lpSupply = pool.strat.totalDeposited();
196         if (lpSupply == 0  pool.allocPoint == 0) {
197             pool.lastRewardBlock = block.number;
198             return;
199         }
200
```

```
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 sYSLReward = (multiplier * sYSLPerBlock * pool.allocPoint) /
            totalAllocPoint;

        if (block.timestamp - lastTeamUpdate >= 1 days && teamRate > 0) {
            address teamWallet = 0xa7b9fFa1BB64856AA62C762bBB0a2A6792DF9613;
            lastTeamUpdate = block.timestamp;
            IsYSL(sYSL).mintFor(teamWallet, teamRate);
            sYSLReward -= teamRate;
        }

        pool.accsYSLPerShare = pool.accsYSLPerShare + ((sYSLReward * DECIMALS) /
            lpSupply);
        pool.lastRewardBlock = block.number;
    }
```

Listing 3.6: `YSLProtocol::updatePool()`

Recommendation Revisit the team rewards design so that it is fairly applied to current pools.

Status The issue has been addressed by taking the above suggestion to improve the team reward allocation.

3.7 Possible Sandwich Attacks in sYSLShare

- ID: PVE-007
- Severity: High
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As part of the protocol logic, there is a constant need to convert one token to another. And the current protocol is designed to interact with various `UniswapV2` pools for token conversion. Our analysis shows that the conversion can be improved by specifying effective slippage control to avoid unnecessary loss.

```
    function transferIn(address _user, uint256 _amount) public override onlyRole(
        STRAT_ROLE) returns (uint256) {
        IStrategySwap(stratSwap).migrate(_amount, address(this));
        IERC20(BUSD).safeTransferFrom(stratSwap, address(this), _amount);

        address _YSL = IStrategySwap(stratSwap).YSL();
        address sYSL = IStrategySwap(stratSwap).sYSL();

        // Get entrance fee from incoming BUSD
```

```
        uint256 fee = _amount / 10;
        _amount -= fee;

        IERC20(BUSD).safeTransfer(feeAddress, fee);

        // Calculate YSL emissions and add it together with BUSD to locked liquidity
        uint256 emission = getPrice(BUSD, _YSL, IERC20(BUSD).balanceOf(address(this)),
            apeSwap);
        IsYSL(_YSL).mintFor(address(this), emission);
        addLiquidity(BUSD, _YSL, IERC20(BUSD).balanceOf(address(this)), emission,
            apeSwap, adapter);

        // Calculate sYSL emission equivalent for BUSD
        uint256 sYslEmission = getPrice(BUSD, sYSL, _amount, apeSwap);
        IsYSL(sYSL).mintFor(address(this), sYslEmission);

        // Add emission
        userDeposited[_user] += sYslEmission;
        totalDeposited += sYslEmission;
        return sYslEmission;
    }
```

Listing 3.7: `sYSLShare::transferIn()`

To elaborate, we show above the `transferIn()` routine from the `sYSLShare` contract. We notice the `sYSL` token is minted based on the calculated `YSL` emissions via `getPrice()`. 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!

```
    function getPrice(
        address token0,
        address token1,
        uint256 amount,
        address _router
    ) internal view returns (uint256 out) {
        address[] memory path = new address[](2);
        path[0] = token0;
        path[1] = token1;
        out = IPancakeRouter02(_router).getAmountsOut(amount, path)[1];
    }
```

Listing 3.8: `YSLOpt::getPrice()`

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, including `processCollectedFee()` and `collectxYSLFee()` from the `xYSLAdapter` contract, as well as `getBusdAmount()` from the `StrategySwap` contract

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

Status This issue is addressed by following the above suggestion to add necessary slippage to mitigate or even prevent the sandwich attack.

4 Conclusion

In this audit, we have analyzed the design and implementation of the YSL.IO protocol. The audited system presents a unique addition to current DeFi offerings by optimizing and amplifying the returns from yield farming platforms through the maximization of locked liquidity. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

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

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

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